Copyright
by
Varun Srivastava
2010

Static Analysis for Finding Security Inconsistencies Between Similar Implementations

by

Varun Srivastava, B.Tech

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2010

The Thesis Committee for Varun Srivastava

Certifies that this is the approved version of the following thesis:

Static Analysis for Finding Security Inconsistencies Between Similar Implementations

APPROVED BY

SUPERVISING COMMITTEE:

Kathryn S. McKinley, Co-Supervisor

Vitaly Shmatikov, Co-Supervisor

Michael D. Bond

Dedicated to my parents.

Acknowledgments

This thesis represents joint work with Prof. Vitaly Shmatikov, Prof. Kathryn S. McKinley, and Dr. Michael D. Bond. My contributions are the proposing, design, implementation, and evaluation of the algorithm for differencing similar implementation of APIs to retrieve security vulnerabilities. I would like to thank my supervisors Vitaly and Kathryn for advising this research. This accomplishment would not be possible without the valuable advice from Michael.

Lastly and most importantly I would like to thank my parents, my dear sister and friends for their selfless love and support.

Static Analysis for Finding Security Inconsistencies Between Similar Implementations

Varun Srivastava, M.A.
The University of Texas at Austin, 2010

Supervisors: Kathryn S. McKinley
Vitaly Shmatikov

The proliferation of distributed, multilayer software services is encouraging a separation of Application Programming Interfaces (APIs) and their implementation, and thus multiple implementations of the same API. Increasing number of platforms are following the Software As A Service (SAAS) model [2, 18, 20, 31], which encourages multiple implementations of the same functionality. To work securely and seamlessly on top of these platforms, software applications rely on consistent implementations of APIs. Vulnerabilities, or interoperability bugs due to differences in security semantics in these APIs, can be exploited to break the security of applications using them. Previous techniques for finding security vulnerabilities and verifying security properties, require manually provided security specifications, which limits their scope [9, 11, 19]. Techniques which automatically extract security policies tend to have a large number of false positives [41].

This work proposes a novel method for automatically extracting security policies and then differencing them to exploit multiple implementations of the same functionality to find errors. We perform context-sensitive, interprocedural forward dataflow analysis to extract the security policies from each implementation and difference them. Determining which security policy is *correct* is difficult. Instead, we exploit the fact that multiple implementations of the same API should have *consistent* security semantics, i.e., we do not determine which one is correct, but which one(s) are different.

We compare the Sun, Harmony and Classpath Java Virtual Machine libraries using our approach and produce very encouraging results. Our approach finds 15 unique cases of security-relevant semantic differences (manifested in 46 APIs) between Sun and Harmony, and 18 cases of security-relevant semantic differences (manifested in 303 APIs) between Sun and Classpath. All these semantic differences are either exploitable vulnerabilities, or bugs resulting in interoperability issues. The approach is effective for accurately finding security vulnerabilities. It takes advantage of the fact that multiple implementations of APIs should have the same security semantics.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **xii**

List of Figures **xiii**

Chapter 1. Introduction **1**

Chapter 2. Background **7**

2.1 Java Runtime Environment 7

2.2 Test Subject Programs 8

2.2.1 Sun Java 1.6.2 8

2.2.2 Harmony 9

2.2.3 Classpath 9

Chapter 3. Related Work **10**

3.1 Static Policies 10

3.2 Dynamic Policies 11

3.3 Program Differencing 13

Chapter 4. System Overview **15**

Chapter 5. Motivation for Static Analysis in Our Implementation **20**

5.1 Interprocedural Forward-Flow MUST and MAY Analysis . . . 20

5.2 Context-Sensitive Interprocedural Constant Propagation and Relational Expressions Evaluation 24

Chapter 6. Algorithm and Implementation **26**
6.1 Static Analysis 26
6.1.1 Dataflow Analysis 26
6.1.2 Compiler Equations 27
6.2 Extracting Security-Relevant Semantic Differences 32
6.2.1 Pattern Extracting Algorithm 33
6.3 Reporting Security Semantic Differences 34

Chapter 7. Evaluation and Results **35**
7.1 Detected Semantic Differences 36
7.1.1 Sun vs Harmony 36
7.1.2 Sun vs ClassPath 40
7.2 Limitations 43
7.2.1 False Positives 44
7.2.2 False Negatives 47

Chapter 8. Conclusion **49**

Appendices **51**

Appendix A. Transfer Function Algorithm **52**

Appendix B. Detailed List of Security Related Semantic Differences **56**
B.1 Sun vs Harmony 56
B.1.1 Missing checks in Sun 56
B.1.1.1 Group 1 56
B.1.1.2 Group 2 57
B.1.1.3 Group 3 58
B.1.1.4 Group 4 59
B.1.1.5 Group 5 60
B.1.1.6 Group 6 61
B.1.1.7 Group 7 62
B.1.1.8 Group 8 63

B.1.1.9 Group 9 64
B.1.2 Missing checks in Harmony 66
B.1.2.1 Group 1 66
B.1.2.2 Group 2 67
B.1.2.3 Group 3 68
B.1.2.4 Group 4 69
B.1.2.5 Group 5 69
B.1.2.6 Group 6 70
B.1.2.7 Group 7 71
B.1.2.8 Group 8 71
B.1.3 False positives eliminated due to interprocedural constant propagation and relational expression folding in Sun 72
B.1.3.1 Group 1 72
B.1.4 False positives eliminated due to interprocedural constant propagation and relational expression folding in Harmony 73
B.1.4.1 Group 1 73
B.1.4.2 Group 2 73
B.2 Sun vs Classpath 75
B.2.1 Missing checks in Sun 75
B.2.1.1 Group 1 75
B.2.1.2 Group 2 76
B.2.1.3 Group 3 78
B.2.1.4 Group 4 79
B.2.1.5 Group 5 80
B.2.1.6 Group 6 82
B.2.1.7 Group 7 83
B.2.1.8 Group 8 97
B.2.1.9 Group 9 98
B.2.2 Missing checks in Classpath 99
B.2.2.1 Group 1 99
B.2.2.2 Group 2 99
B.2.2.3 Group 3 100

B.2.2.4 Group 4 100
B.2.2.5 Group 5 100
B.2.2.6 Group 6 101
B.2.2.7 Group 7 105
B.2.2.8 Group 8 105
B.2.2.9 Group 9 106
B.2.3 False positives eliminated due to interprocedural constant propagation and relational expression folding in Sun 106
B.2.3.1 Group 1 106
B.2.4 False positives eliminated due to interprocedural constant propagation and relational expression folding in Classpath 107
B.2.4.1 Group 1 107
B.2.4.2 Group 2 108

Bibliography **112**

Vita **118**

List of Tables

7.1 API coverage for different Java Virtual Machines 35
7.2 Sun vs Harmony - Static Analysis Report 37
7.3 Sun vs Harmony - Checks missing in Sun 40
7.4 Sun vs Harmony - Checks missing in Harmony 41
7.5 Sun vs Classpath - Static Analysis Report 42
7.6 Sun vs Classpath - Checks missing in Sun 43
7.7 Sun vs Classpath - Checks missing in Classpath 44

A.1 Security Sensitive Events for the JVM libraries 52

List of Figures

1.1 Java code showing vulnerability in java.net. library of the Harmony and the correct one in the Sun 5

3.1 A typical example of security-sensitive code in Java programs 13

4.1 Main components of our system 16

5.1 Interoperability bug in java.security.KeyStore class. In Sun, security check, checkRead, always dominate the security-sensitive events. 22

5.2 Interoperability bug in java.security.KeyStore class. In Harmony security check dominates security-sensitive events, if the path parameter is not empty. 23

5.3 URL constructor has security check inside a conditional branch. The check will not be performed for the constructor shown above. 25

6.1 Lattice for the forward dataflow analysis. 27

7.1 Extra functionality in Harmony make StreamHandlerPermission necessary for using addURL API 39

7.2 The portion of code present in many APIs dealing with strings in Sun and Harmony 41

7.3 Inappropriate use of security check to check accessible InetAddress 45

7.4 Inappropriate use of security check to differ between hostname and IPv6 Addresses 46

7.5 A hypothetical code segment which can introduce a false negative in our approach. 48

Chapter 1

Introduction

Software development is moving towards the software as a service (SAAS) model on the Internet [2, 18, 20, 31]. With increasing use of cloud computing platforms and proprietary software modules providing specialized functionalities, similar applications are being developed by different vendors, e.g., different Java Virtual Machines or Web browsers. There are also a number of API implementations that provide the same functionality.

Cloud computing encourages development of applications as sets of web service APIs. Each web service API performs a defined task. Applications then combine these web service APIs. Security modules are also being developed as services [14, 21, 35]. The most common access control security model is making security checks, in some specific order, before performing a security-sensitive task. Defining the order of these security checks as a specification is very difficult. Depending upon the task, different sets of security checks are needed. Multi-tiered architectures also have a similar structure, where each tier performs a defined task. Design patterns for these kinds of architectures are becoming more popular with the increasing use of cloud computing and multi-tiered applications.

With the increasing use of memory-safe languages such as Java and C#, and memory-safe systems [8, 40], memory attacks and control flow attacks are becoming less common. Security vulnerabilities are increasingly caused by semantic mistakes in the code [6]. Consistency among different implementations of the same API is required to achieve consistent security policies and coherent program behavior.

Providing a framework for statically checking security policies means providing a complete and sound security specification for each security-sensitive event performed in the application. This type of static analysis has been tried in the past [19]. The major problem with this approach is defining sound and complete security specifications for the analysis. It also relies on the intelligence of the humans providing the security specifications. Typically, the security team provides security specifications to the developers. Differences in understanding between the security team and the development team can lead to subtle semantic bugs. Even when specifications are written by security-aware developers, there is always a great chance of expensive errors.

Other techniques try to extract security semantics from the existing code and then derive some patterns [11, 34]. There is currently no oracle which can classify the rules extracted from the existing codebase as capturing true vulnerabilities or false positives. These approaches extract a huge number of security policies, which result in a large set of false positives.

In our approach, we consider implementations of functionality. Similar functionality can be the same functionality implemented by two different ven-

dors, or it can be as simple as two APIs providing similar functionality. We automatically extract the relation between security checks and security-sensitive events from one implementation, and compare it with other implementations. These relations have to be the same to provide consistent security semantics.

Finding correct security semantics for a system is hard. In the object-oriented programming paradigm, specific modules or classes are responsible for performing all security-related checks. The good news is that multiple implementations of the same API must enforce the same security semantics. A frequent access control paradigm is to check or obtain correct permissions before executing a *security-sensitive event*. For example, the SecurityManager class in Java is responsible for providing Access Control checks. We call these APIs, implementing security-sensitive functionality, *security checks*. These security checks are typically mixed with the application code, to provide the desired security semantics.

For our experiments, we have taken the three popular versions of the Java Virtual Machine libraries. All libraries should have the same security semantics, so that applications developed in Java can run seamlessly on top of any of these libraries. Security semantics for the JVM libraries are hard to define precisely. For example, there is a vulnerability in the Harmony JVM network libraries, where the security check before opening a connection is missing. API openConnection(java.net.Proxy) of java.net.URL class contains a security-sensitive event on a non-public class member.(For the complete list of security-sensitive events see Chapter 6.) The operation is calling method

openConnection(URl, Proxy) over a non-public class member of type URL-StreamHandler. In the Sun JVM library, this action is dominated by a security check "checkConnect", whereas the same action is not dominated by any security check in Harmony.

If we provide a high-level security policy for the code in Figure 1.1, we can say that actions on all non-public class members should be secured by security checks. Since URLStreamHandler is a private data member of class, this policy will perform a security check before it calls openConnection method with the URLStreamHandler variable. Although this security policy will detect the vulnerability in Figure 1.1, it will lead to thousands of false positives, as not all the operations on non-public class members are security-sensitive. Secondly, it will be really hard to define which security check is required for operations on each of the non-public class members. The ideal policy should accurately define the relationship between security checks and the security-sensitive events. Getting such complete and sound security policies are hard. Our objective is to ensure consistency, which is unambiguous. By differencing the security policies, our approach accurately detects this semantic difference in security between Harmony and Sun. The differencing indicates that the openConnection event is not secured by any security check in Harmony, whereas the same event is secured in Sun by the "checkConnect" security check.

We aim to find security-relevant semantic differences between multiple implementations of API. Our approach produces very few false positives. In

```java
//In SUN
public URLConnection openConnection(Proxy proxy){
    ...
    SecurityManager sm = System.getSecurityManager();
    if (proxy.type() != Proxy.Type.DIRECT && sm != null)
    {
          InetSocketAddress epoint =
                  (InetSocketAddress) proxy.address();
          if (epoint.isUnresolved())
              sm.checkConnect(epoint.getHostName(),
                          epoint.getPort());
          else
              sm.checkConnect(epoint.getAddress().
                          getHostAddress(), epoint.getPort
                             ());
     }
     return handler.openConnection(this, proxy);
}

//In Harmony
public URLConnection openConnection(Proxy proxy)
throws IOException {
     ...
     return strmHandler.openConnection(this, proxy);
}
```

Figure 1.1: Java code showing vulnerability in java.net. library of the Harmony and the correct one in the Sun

our experiments, we get most of the false positives due to inappropriate use of security checks for tasks other than access control, as discussed in Section 7.2.1. Security semantics of similar APIs should be identical, therefore any difference points to some issue in one of the API. Our analysis finds security-relevant semantic differences between different JVM libraries, with false positive rate of less than 1 percent. The vulnerabilities are accepted by the respective library implementors [36–38].

Chapter 2

Background

2.1 Java Runtime Environment

Java Runtime Environment (JRE) of Java Virtual Machine (JVM) is often developed in C or C++, whereas utility libraries are developed in Java and native languages, like C. JVM libraries provide networking, security, input and output routines, and many basic language constructs. The public methods of the JVM library can be called by any Java program to perform the desired function. All public and protected methods should be secured properly as they can be accessed by application code. To use services from outside the JVM, applications can use the JNI to access native libraries. Thus, JNI calls and the Java library APIs are the entry points to security sensitive events. We restrict our attention to the Java libraries.

Java security relies on sandboxing for its access control model [30]. SecurityManager class APIs are primarily responsible for enforcing Java sandboxing. If a Java program is executed inside the Java sandbox, then SecurityManager checks are performed. The Java 2 platform security model extends security policies by incorporating new security policy classes. Security check can be performed inside an application by simply calling checkPermission with

an appropriate permission type.

Different implementations of JVM libraries should have the same semantics in order to provide interoperability for Java applications. All JVM libraries should follow the Sun Java Compatibility guidelines and pass a set of tests [29]. However, these guidelines and tests do not fully capture the entire security semantics of the JVM libraries. Differences in security semantics of different implementations of JVM libraries can lead to loss of interoperability and expose the system to exploitable security vulnerabilities.

2.2 Test Subject Programs

As a proof of concept we evaluated the security semantics of JVM libraries developed by Sun [28] and open-source Harmony [3] and Classpath [16] which are, respectively, supported by IBM, Apache, and several other companies and open-source developers.

2.2.1 Sun Java 1.6.2

The Sun JVM library implementation is the most widely used implementation [28]. Java language was developed by Sun, and thus the functional specifications for the libraries are provided by Sun, which must be followed by all the implementations. It is desirable that programs developed on one implementation of the JVM libraries will also work seamlessly on all other JVMs as well.

2.2.2 Harmony

The Harmony JVM libraries are open-source JVM library implementation [3]. Harmony is used in the implementation of the IBM JVM libraries and in Google's Android Dalvik virtual machine libraries [17].

2.2.3 Classpath

Classpath is another open-source JVM implementation which is supported by many open-source developers and companies [16]. Some of the Classpath based systems are:

- *IKVM* - A JVM for the .NET platform,which executes on top of Mono [27].
- *Kaffe* - Kaffe virtual machine runs on top SuperH, IBM z series mainframes and PlayStation 2 [23].
- *JNode* - JNode is a Java New Operating System Design Effort [22]. The goal is to get a simple to use and install Java operating system for personal use.
- *GCJ* - GCJ is the GNU compiler for the Java programming language. A lot of work has been done to merge GCJ and Classpath class library code [15].

Chapter 3

Related Work

We divide the related work section into 3 categories: static, dynamic and program differencing policies.

3.1 Static Policies

Dillig et al. find bugs caused by inconsistent checks done in the program [9]. For example, if in one place we have if(x!=NULL)*x; and later *x=y; is present, then we can neither assume x to be NULL, nor non-NULL. Their approach extracts these kinds of conflicting assumptions. The focus of the work is on null-pointer dereferences. Moreover, it requires contradictory policies to extract semantic differences. The approach requires a well-defined security policy for each kind of vulnerability. Coming up with these policies can be onerous and tedious.

Kim et al. keep a database of previously detected bugs [24]. They scan new code hunks for the old bug patterns to detect the reoccurrence of already fixed bugs. This approach helps in finding bugs in the new code only, and only if a similar bug was fixed before.

FindBugs tool by Hovemeyer and Pugh takes a set of patterns described

by the programmer and then tries to find whether or not there are vulnerabilities matching the pattern provided [19]. The tool requires patterns to be provided by the programmer [12]. These patterns are textually matched in the code.

Sistla et al. [32] perform model checking on JVM libraries. They define security-sensitive events, which should be secured by SecurityManager checks. The approach defines only calls to native methods as security-sensitive events. Unambiguously defining security-sensitive events is hard as the sensitivity of events depends upon the context in which they are performed. Thus with other Java security-sensitive events, the approach will generate many false positives. The SLAM and MOPS projects [5, 7] apply a similar approach for C programs.

3.2 Dynamic Policies

Engler et al. try to automatically extract the templates from a given program [11]. This approach categories 'beliefs' into MAY and MUST belief groups. MUST beliefs are the ones which should be always true. For example, if a pointer is dereferenced, then is should not be null, a call to unlock should be preceded by lock on the same object. MUST beliefs are generally provided by the programmer. MAY beliefs are the ones which are not concretely defined in the program. MAY beliefs are highly dependent on the semantics of the program. For example, a call A should always be dominated by another call B can be a MAY belief. The approach tries to retrieve templates from the given program and then apply it to the same program. For obtaining valid beliefs

from huge set of MAY beliefs, the templates which occur more often are given preference. The important events which occur rarely in the programs will not be captured by this approach, as there will not be many instances to increase the weight of the event. In JVM libraries, a particular security check guards different security-sensitive events in different parts of the code. MAY beliefs extracted from this type of code will not be too useful as there will not be many instances to support the correct code against the vulnerable code.

Tan et al. extract templates for security checks on data structures accesses [34]. Their approach extracts templates by looking at the security calls which dominate the important data structures of Linux kernel. The security checks which occur frequently are preferred over rarely occurring security checks as in the case of Engler et al [11]. Their approach has same shortcomings as Engler's [11] MAY template cases. Moreover, the approach only tracks whether important data structures are secured by security checks or not. It does not look for security-sensitive API calls which should also be dominated by security checks. For example, in JVMs some native calls must be secured by security checks. The approach only looks for syntactic domination of security-sensitive data structures. Tan does not perform context-sensitive interprocedural dataflow analysis to find must or may domination of security checks over security sensitive data structures.

Whaley et al. extract the finite-state-machine based submodels of the interface of a class [39]. Whaley builds the FSM based on the memory access of common data structures. Their approach will not extract the basic secu-

```
sm.checkPermission();
doSensitiveOperation();
```

Figure 3.1: A typical example of security-sensitive code in Java programs

rity model we examine. For example, it does not capture the property that doOperation() should be dominated by chechPermission(), for the code given in figure 3.1, because there is no data structure shared between checkPermission() and doOperation(). The approach only models operations which have some data structure in common. Even in the case of operations where data structures are shared, the approach does not provide context sensitivity.

Ganapathy et al. use concept analysis to extract security-sensitive operations [13]. Their approach is complementary to ours. In our case, we provide high-level definition of security-sensitive events as input. Using their approach, we can automatically define security-sensitive events.

3.3 Program Differencing

Lots of research has been done on clone detection, focusing on extraction of semantic and syntactic differences between similar code sections of a program. Techniques based on textual and token [4, 10] extraction will not work across different implementations, as different implementations can use different algorithms and data structures to achieve same functionality. Some clone detection techniques [25, 26] try to capture semantic behavior of programs. Some of these techniques may be used in our tool to capture semantic

behavior across implementations. Extracting effective semantic behavior depends upon the system being analyzed. In our experiments, we have proposed an effective technique to extract *security-relevant* semantic behavior of applications using access control.

Chapter 4

System Overview

This section describes our approach for extracting vulnerabilities and interoperability bugs between different implementations of similar APIs.

Multiple implementations of similar APIs must enforce the same security semantics. The differences in security semantics indicate a security vulnerability or interoperability issues. The Java security model uses the access control paradigm, which involves security-sensitive events performed after proper security checks. We extract access-control policies from one implementation of the JVM libraries and then difference them with the other implementations to detect the semantic differences in policies.

Figure 4.1 shows the division of our approach into 6 main steps. We now overview the six steps in our approach.

1. **Find Similar APIs**: We examine different implementations of the same public and protected APIs in multiple JVM libraries. We compare the implementations of the methods, which have the same signature.

   In our experiments, similarity is based on closeness in functionality, i.e., we analyze different implementations of the JVM libraries performing the same task. Different criteria for deciding the similar methods are

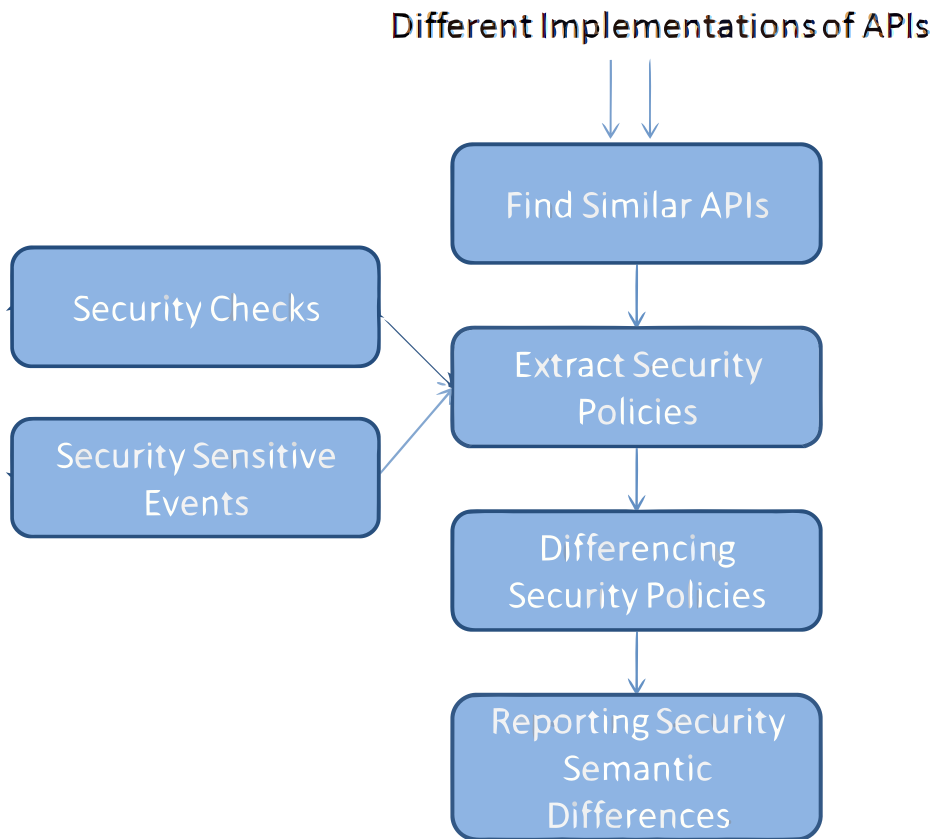

Figure 4.1: Main components of our system

also possible. Similarity between methods can depend on data structures usage or security checks. Finding good criteria for deciding which methods are similar remains an open research problem.

2. **Security Checks or Templates**: Security checks are the set of method calls, responsible for providing access-control checks in the system. We provide set of security check APIs as an input to the system. We do not expect detailed static templates [19]. In our experiments, we look for a set of security calls dominating some security-sensitive events. For example, inside JVM libraries, the *java.lang.SecurityManager* package is resposible for implementing the reference monitor. We treat calls to SecurityManager APIs as security checks.

3. **Security-Sensitive Events**: Likewise, we use a very broad definition of security-sensitive events, which are protected by security checks, to cover as many events as possible. In our system, our aim for the definition of security-sensitive events is soundness. False positives are always an issue with sound analysis. Differencing of security policies in later steps takes care of false positives. For our experiments, we define the security-sensitive events as something that might need to be protected because it can affect the JVM or state outside the application, such as network activity or file I/O. Section 6 gives a detailed list of security sensitive events.

4. **Extracting Security Policies**: Security policies are sets of security-

sensitive events dominated by security checks. Since our system extracts high-level security checks and security-sensitive events, it produces a huge number of security policies for each API. In our experiments, we track calls to SecurityManager APIs, which are responsible for making JVM libraries secure, and discover all security-sensitive events they should protect. We extract two types of policies:

- *MUST*: Code must always enforce these policies. In these policies security-sensitive events are always dominated by a certain security check.
- *MAY*: On some control paths, the policy is enforced. In these policies, security-sensitive events are dominated by a security check depending upon certain conditions. For example, in code shown in Figure 5.2, 'checkRead' security check depends upon the length of 'path' variable.

5. **Differencing Security Policies**: To extract inconsistencies, we difference security policies of one implementation, with the policies of a different implementation of the API. For example, in Figure 1.1 the SUN implementation calls openConnection method after performing 'checkConnect' security check. In Harmony, the same security-sensitive event is not dominated by any security check. By differencing these two policies, we infer, that the security-sensitive event openConnection is protected in SUN but not in Harmony. The differencing of security policies of differ-

ent implementations of the same API, gives a list of interesting semantic differences, each of which corresponds to a security vulnerability in one of the implementations, or an interoperability bug.

6. **Reporting Semantic Differences**: Finally we report these MUST and MAY policy differences. The semantic differences found between different implementations fall in the following three categories.

   - *Vulnerability*: A semantic difference which can be easily exploited in one of the implementations to perform some action which should be prohibited.

   - *Interoperability Bugs*: A semantic difference which causes interoperability problems. Prima facie these semantic differences do not give the attacker power to perform undesirable actions. These kind of harmless-looking semantic differences can be used, however, to stage a complex multi-stage attack [6].

   - *False Positives*: A semantic difference which is reported due to imprecision of our approach, or the one which does not results in the difference in security related semantics, is classified as a false positive. For example in our experiments, we had cases where security checks were improperly used for exception handling in one implementation of JVM libraries as shown in Figure 7.3.

Chapter 5

Motivation for Static Analysis in Our Implementation

In this chapter, we motivate our choice of the static analysis algorithms. Our static analysis evolved incrementally with the challenges we faced while analyzing JVM libraries.

5.1 Interprocedural Forward-Flow MUST and MAY Analysis

We extract security policies by looking at security-sensitive events protected by security checks. More precisely, we find whether security-sensitive events are dominated by security checks or not. We perform forward dataflow analysis to find MAY and MUST domination of security checks over security-sensitive events. The security-sensitive events may be part of one method, whereas security checks may be performed inside another. To find the relation between them, we perform interprocedural forward-flow analysis. The analysis gives us interprocedural MAY and MUST dominance of security checks over important events.

We now show an example taken from the Sun, Figure 5.1, and the

Harmony JVM libraries, Figure 5.2, which requires interprocedural, MAY and MUST forward dataflow analysis. In newInstance method of the Sun JVM library, security check “checkRead” MUST dominate all the security sensitive events, whereas in Harmony the security check is done only if the path parameter is not empty.

The API newInstance(String,Provider,File,ProtectionParameter) in class KeyStore of java.security library takes File as an argument. Both Harmony and Sun check whether the File argument is a file link or not. In Harmony, inside isFile() method, first it checks the length of the path. If the supplied path is an empty string, the method returns false. If the path has some value, then the method performs the checkRead() security check. The checkRead() security check is therefore optional in Harmony, i.e., it is done only if the path of the file is non-empty. The empty path check is not performed in Sun, and hence the security check, checkRead(), is MUST for all events happening after the call to the isFile() method. This semantic difference will change behavior. If the file path parameter is empty, then in Sun it will result in a security exception, as there will be no security policy corresponding to the empty path in the normal case. In Harmony, the empty path check will return false from the isFile() method, resulting in an IllegalArgumentException. The checkRead() security check is performed inside the isFile() method, whereas security sensitive events such as returning an object of type java.security.keystore.Builder are performed inside the newInstance() method. Capturing this difference in behavior requires interprocedural MAY/MUST dataflow analysis.

```
//In Sun
public static Builder newInstance(String type,
Provider provider, File file, ProtectionParameter
   protection)
{
        ...
 if (file.isFile() == false) {
   throw new IllegalArgumentException(...);
 }
 return ...;
}

public boolean isFile()
{
 SecurityManager security = System.getSecurityManager();
 if (security != null) {
  security.checkRead(path);
 }
 return ...;
}
```

Figure 5.1: Interoperability bug in java.security.KeyStore class. In Sun, security check, checkRead, always dominate the security-sensitive events.

```
//In Harmony
public static Builder newInstance(String type,
Provider provider, File file,
ProtectionParameter protectionParameter)
{
    ...
if (!file.isFile()) {
 throw new IllegalArgumentException(...);
}
return ...;
}

public boolean isFile() {
 if (path.length() == 0) {
  return false;
 }
 SecurityManager security = System.getSecurityManager();
 if (security != null) {
  security.checkRead(path);
 }
 return ...;
}
```

Figure 5.2: Interoperability bug in java.security.KeyStore class. In Harmony security check dominates security-sensitive events, if the path parameter is not empty.

5.2 Context-Sensitive Interprocedural Constant Propagation and Relational Expressions Evaluation

We perform interprocedural constant propagation and evaluate basic relational expressions, consisting of null check for objects and integral constants to improve the precision of our analysis. To track constants and security-sensitive variables interprocedurally we perform context-sensitive analysis. We traverse each path of the call graph separately. This exhaustive traversal of call graph makes our analysis naturally context-sensitive.

We eliminate false positives by removing unreachable code containing security checks. For example in Figure 5.3, consider the constructor of java.net.URL, URL(URL, String, URLStreamHandler handler). If the handler object is not null, then the constructor performs a check for the "specifyStreamHandler" Permission. In the example, URL(String spec) constructor calls this constructor with handler passed as null. In this case, the check for "specifyStreamHandler" Permission will never be called. To capture this case, we need interprocedural constant propagation and evaluation of simple relational expressions as described in Section 6. This case also requires context-sensitive dataflow analysis because the same URL Constructor can be called with different arguments at different places.

```
// In Harmony
public URL(String spec) throws MalformedURLException
{
 this((URL) null, spec, (URLStreamHandler) null);
}
public URL(URL context, String spec,
URLStreamHandler handler)
 throws MalformedURLException
 {
 if (handler != null) {
  SecurityManager sm = System.getSecurityManager();
 if (sm != null) {
  sm.checkPermission(
  specifyStreamHandlerPermission);
  }
 strmHandler = handler;
 }
 ...
}
```

Figure 5.3: URL constructor has security check inside a conditional branch. The check will not be performed for the constructor shown above.

Chapter 6

Algorithm and Implementation

This chapter describes our algorithm and implementation details.

6.1 Static Analysis

We use the Soot Compiler Analysis framework to implement our analysis. Data Flow analysis is performed after converting class files of the JVM libraries into a Soot intermediate representation Jimple. Jimple is the typed, 3-address, statement-based intermediate representation of the Java programs.

We design and implement a Forward-Flow Analysis module, called *RegalTransformer*, as an extension of BodyTransformer in the Soot framework. Body Transformers get the full method body as input to perform dataflow analysis.

6.1.1 Dataflow Analysis

We perform context-sensitive, interprocedural forward dataflow analysis on all the public and protected methods present inside the JVM library. We analyze public methods because they can be called directly by any user. We also analyze protected methods because any user can create a subclass

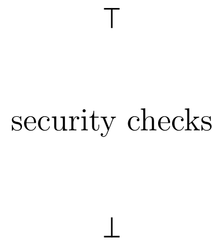

Figure 6.1: Lattice for the forward dataflow analysis.

of the class, containing the unsecured protected method, and then invoke the unsecured method from any method inside the subclass.

6.1.2 Compiler Equations

For forward dataflow analysis, we define TransferFunction over each Java statement 'v'. The method which is getting analyzed is called the *root method*.

TransferFunction TransferFunction(v) generates security policies, i.e., security-sensitive events tagged with appropriate security checks. TransferFunction returns the list of security checks called inside the statement 'v'. The detailed TransferFunction algorithm is given in Appendix A.

Security-Sensitive Events While analyzing an API, we look for the following security-sensitive events.

1. We track operations performed on all non-public declared class members, which define the state of the object. Therefore, any modification done

on these variables is an important part of the functionality performed by the method.

2. We distinguish calls to native methods as security-sensitive events. Native methods are responsible for the interaction of the JVM with the outside environment.

3. We analyze operations on arguments passed to the root method. Among the different implementations of the same API, the arguments passed to the root method are always same, therefore the operation performed on these arguments should be the same.

4. We also analyze returns. The value finally returned from the API is also security-sensitive. Many APIs are used to extract critical data from the system. For example, System.getProperty(String) method is used to retrieve the properties of the system. This critical information is only returned if the code has permission to retrieve the data. Therefore, it is important to track return statements.

Security Checks Security checks are the set of APIs which are responsible for performing the access control checks before the security-sensitive events. For JVM libraries, these are the methods present in the java.security.SecurityManager class.

The aim of our dataflow analysis is to determine which security-sensitive events are MUST or MAY dominated by the security checks responsible for im-

plementing the reference monitor in the JVM libraries. We perform following forward dataflow analysis:

- *MAY dataflow analysis*: The aim of MAY dataflow analysis is to generate MAY security policies. As shown in Algorithm 1 the MAY dataflow analysis starts by assigning $\perp$ to the set of security checks leaving each statement. The worklist is initialized with the EntryPoints of the root method. The set of security checks reaching the statement 'v' is calculated by taking union of all the sets of security checks leaving the predecessors of the statement 'v'. The lattice shown in Figure 6.1 is used for the merge opeartions. The TransferFunction(v) returns the set of security checks performed inside statement 'v'. If no security check is performed in a statement, then the TransferFunction returns $\perp$. The set of security checks leaving the statement 'v' is constructed by taking union of the value returned by TransFerFunction(v) with the set of security checks reaching the statement 'v'. If the statement 'v' is analyzed for the first time or, if the set of security checks leaving the statement 'v' changes, we add the successors of the statement 'v' in the worklist. The algorithm keeps going until we have statements in the worklist.

- *MUST dataflow analysis*: The aim of MUST dataflow analysis is to generate MUST security policies. MUST dataflow analysis starts by assigning $\top$ to the set of security checks leaving each statement. MUST dataflow analysis progress exactly like MAY analysis, just the set of security checks reaching the statement 'v' is calculated by taking intersection

of all the sets of security checks leaving the predecessors of the statement 'v'. The MUST analysis is shown in Algorithm 2.

Algorithm 1 Compiler equation for MAY dataflow analysis

EntryPoint is the start of the method.
ExitPoints are the return statements.
for all $v \epsilon ALLSTATEMENTS$ **do**
  $OUT(v) \leftarrow ^{\perp}$
  $NOTVISITED \leftarrow NOTVISITED \cup v$
end for
$worklist \leftarrow EntryPoint;$
while $worklist \neq \phi$ **do**
  $v \leftarrow worklist.getNode()$
  $IN(v) \leftarrow \bigcup(OUT(p)), p\epsilon PRED(v)$
  $OUT(v) \leftarrow IN(v) \cup TransferFunction(v)$
  if $OUT(v)changed || v\epsilon NOTVISITED$ **then**
    $worklist \leftarrow worklist \cup SUCC(v)$
    $NOTVISITED \leftarrow NOTVISITED - v$
  end if
end while

Our approach analyze all the public and protected methods of JVM libraries one by one. The list of security-sensitive events with the security checks dominating them is recorded in a hash table, indexed by the root method signature. We call these lists of security-sensitive events with security checks dominating them, the *security policies* of the method.

JVM libraries are supposed to work with and without the presence of Security Manager security checks. If an application is running on a standalone machine, then it may not require a secure environment. Therefore, all security checks inside the JVM libraries are performed if the security manager

Algorithm 2 Compiler equation for MUST dataflow analysis

EntryPoint is the start of the method.
ExitPoints are the return statements.
for all $v \epsilon ALLSTATEMENTS$ **do**
  $OUT(v) \leftarrow \top$
  $NOTVISITED \leftarrow NOTVISITED \cup v$
end for
$worklist \leftarrow EntryPoint;$
while $worklist \neq \phi$ **do**
  $v \leftarrow worklist.getNode()$
  $IN(v) \leftarrow \bigcap (OUT(p)), p \epsilon PRED(v)$
  $OUT(v) \leftarrow IN(v) \cup TransferFunction(v)$
  if $OUT(v) changed \, || \, v \epsilon NOTVISITED$ **then**
    $worklist \leftarrow worklist \cup SUCC(v)$
    $NOTVISITED \leftarrow NOTVISITED - v$
  end if
end while

is present. For MUST dataflow analysis, security checks will never MUST dominate security-sensitive events. To get rid of this situation in MUST dataflow analysis, we assume the SecurityManager is always present and modify MUST analysis to take care of this situation when merging different data flow branches.

When performing interprocedural dataflow analysis, the forward-flow analysis algorithm described above is performed on each root method. At the end of the analysis, if the root method had at least one security call in the full call path, then we store security policies, which are the security checks dominating security-sensitive events; otherwise we do not store anything for the method. The security-sensitive events not dominated by any security call will be reported, when we difference the extracted security policies from

different implementations. We do not store any data for the methods, that have no security checks, which is the most common case.

We aim to extract precise security policies. To achieve this goal, we need to eliminate unreachable code segments. We explore each path of the call graph seperately. This makes our analysis naturally context sensitive and enable us to perform interprocedural constant propagation. Exploring the call graph exhaustively makes our analysis expensive in terms of computation time. Achieving the desired context information without exhaustive call graph traversal will be the focus of our future work. We perform constant expression folding involving relational operators. If the relational expression evaluates to a constant, we skip the unreachable branch. The interprocedural constant propagation makes the expression folding more effective. The interprocedural constant propagation and expression folding involving relational operators, help us in reducing false positives as discussed in Section 5.2.

6.2 Extracting Security-Relevant Semantic Differences

After collecting security policies from the JVM libraries, as described in Chapter 6.1, we feed data to our Data Analysis module. The Data Analysis module fetches methods with the same signature from different implementations and compares their security policies.

Between different implementations of JVMs libraries, we detect semantics differences as opposed to extracting the entire generic policy [39].

Data Analysis module reports following semantic differences for similar security-sensitive events across different implementations:

- Security-sensitive events in one implementation are either MAY or MUST dominated by a security check, whereas the other implementation does not have that security check dominating the same event. Figure 1.1 shows an example.

- Security-sensitive events in one implementation MAY dominated by a security check, whereas the other implementation has the same security check MUST dominating the same event. Figure 5.2 shows an example.

To extract security-semantic differences, we introduce a Pattern Extracting Algorithm.

6.2.1 Pattern Extracting Algorithm

We take the security policies of the same method from two different implementations. Security policies are generated for each method as described above. We report a semantic difference under the following conditions.

- *if the first method does not have any security policy, whereas second has a list of security policies* - These security policies clearly differ.

- *if both the methods have list of security policies* - Then we compare the policies for both MAY and MUST. This test generates reports when

security sensitive events are either missing some security checks in one implementation, or there is a mismatch in the MAY or MUST policies.

6.3 Reporting Security Semantic Differences

We then report these security-related semantic differences. To help users understand the results better, Data Analysis automatically group the APIs according to the places where security checks are missing or improper. Our summaries help users understand the root cause of semantic differences by decreasing the size of the difference report. For example, a security check missed in some character decoder will be reflected in all string APIs and the APIs which use the string APIs. By grouping the methods according to the places where the security check is missing or inappropriate, we can quickly analyze the final report generated by the tool. So in our example, we can see all the JVM libraries showing difference in semantic security behavior due to the missing check in character decoder under one group.

Chapter 7

Evaluation and Results

We evaluated our tool on the following versions of the JVM libraries

1. Sun Java Virtual Machine libraries 1.6.0_07

2. Apache Harmony libraries 1.5.0 svn revision r761593

3. GNU ClassPath libraries version 0.97.2

We analyze and compare about 3M lines of code.

Table 7.1 shows the coverage of our analysis. We analyze 8013, 6572 and 21281 methods for Sun, Harmony and Classpath respectively. We perform MAY and MUST dominance of security checks analysis over security-sensitive events.

In the rest of the chapter we will discuss the vulnerabilities and interoperability bugs we found in different JVM libraries. We will also discuss the

	Sun	Harmony	Classpath
Number of Methods	8013	6572	21281
Number of Methods having security calls	421	237	2579
Time for MAY analysis(mins)	300	190	340
Time for MUST analysis(mins)	560	290	650

Table 7.1: API coverage for different Java Virtual Machines

false positives and false negatives for our approach.

7.1 Detected Semantic Differences

In our experiments, we compare Sun JVM libraries against Harmony and ClassPath JVM libraries. Sun JVM libraries are the most widely used, therefore, comparing other JVM libraries against Sun is logical. In our experiments, we find that not only ClassPath and Harmony JVM libraries contain vulnerabilities, by comparing to Sun, but the Sun libraries also contain vulnerabilities. All of these vulnerabilities and inconsistent security semantics will cause the same Java code to behave differently depending on which JVM library is used. The vulnerabilities reported by our analysis are accepted as bugs by the respective library implementors [36–38].

7.1.1 Sun vs Harmony

Between Sun and Harmony, we find 49 security semantic differences. Table 7.2 summarizes the result of static analysis performed for Sun and Harmony. Sun is missing security checks in 23 cases, whereas Harmony is missing security checks in 18 cases. Five cases are there due to a mismatch in MAY and MUST domination of same security checks. Three false positives are also reported. The false positives are due to inappropriate coding practices used in Harmony JVM libraries as described in Section 7.2.1.

Number of Methods present both in Sun JVM and Harmony	4447
Number of Methods having security check in both Sun and Harmony	199
Number of Methods having security check in Sun and present in Harmony	217
Number of Methods having security check in Harmony and present in Sun	225
Number of Methods having security check in Harmony, but no security check in Sun	31 (9 distinct groups)
Number of Methods having security check in Sun, but no security check in Harmony	18 (8 distinct groups)
Number of differences found due to Intraprocedural Analysis	6 (5 groups)
Number of differences found due to Interprocedural Analysis	38 (11 groups)
Number of differences found due to Difference in May/Must check	5 (1 group)
Number of False positives reported between Sun and Harmony	3 (2 group)
Number of False positives eliminated due to Interprocedural Constant Propagation	35 (4 groups)

Table 7.2: Sun vs Harmony - Static Analysis Report

Sun Semantic Differences In the Sun JVM libraries, we have 2 cases of exploitable vulnerabilities. We reported these bugs and Sun accepted them as bugs [38]. The 21 other cases of security semantic differences in Sun can be classified into Interoperability Bugs. These are extra security checks performed in Harmony due to some features not present in Sun. Although these security checks may not be required for Sun, they still create interoperability issues. These APIs in Harmony JVM libraries will require extra permissions to execute in comparison to Sun. Figure 7.1 shows an example where extra functionality in Harmony makes StreamHandlerPermission necessary for using the addURL API of the URLClassLoader class. The permission is not required for the addURL API in the Sun JVM library.

Table 7.3 depicts the nature of security semantic differences found in the Sun APIs in comparison to Harmony. We put the semantic differences that have the same root cause in a single group. All semantic differences result in a difference in security behavior, but in case of vulnerabilities, they can be easily exploited and thus should be immediately fixed.

Harmony Semantic Differences Table 7.4 depicts the nature of security semantic differences found in Harmony APIs in comparison to Sun.

In the Harmony JVM libraries, we have 10 cases of exploitable vulnerabilities. We reported these errors and they were accepted by Harmony [37]. The 8 other cases are security semantic differences, which can be classified as Interoperability Bugs. These are extra security checks performed in Sun due

```
//In Harmony
URLClassLoader:void addURL(URL url) {
...
createSearchURL(url);
...
}
 URL createSearchURL(URL url) {
  ...
   if (isDirectory(url) || protocol.equals("jar")) {
           return url;
       }
       if (factory == null) {
           return new URL("jar", "",-1, url.toString()
              + "!/");
       }
       return new URL("jar", "", -1, url.toString() + "
          !/",
              factory.createURLStreamHandler(protocol)
                 );
  ...
 }
 //URL constructor performs
 //checkPermission(specifyStreamHandlerPermission)
 //security check for the stream handler passed.
 //whereas in Sun there is no such functionality

 // In Sun
 protected void addURL(URL url) {
       ucp.addURL(url);
  }
```

Figure 7.1: Extra functionality in Harmony make StreamHandlerPermission necessary for using addURL API

Group	# of APIs in group	Analysis Required	Differences
1	1	Intraprocedural	Semantic difference
2	6	Interprocedural	Semantic difference
3	2	Interprocedural	Vulnerability in Sun
4	1	Intraprocedural	Semantic difference
5	12	Interprocedural	Semantic difference
6	1	Interprocedural	Semantic difference
7	5	Interprocedural	Semantic difference
8	1	Interprocedural	False Positive
9	2	Interprocedural	False Positive

Table 7.3: Sun vs Harmony - Checks missing in Sun

to some extra features not present in Harmony. These security checks may not be required for Harmony, but still they create interoperability issues. For example, API java.lang.String: byte[] getBytes() misses the 'checkExit' check present in Harmony as shown in Figure 7.2. This interesting semantic difference between Harmony and Sun which arises due to the difference in the way they react to an Exception. In Sun, if the default "ISO-8859-1" charset decoder is not present, then it terminates the application by calling System.exit(), while Harmony throws the UnsupportedEncodingException. To perform System.exit(), the user requires Exit permission, and hence Sun has checkExit check inside System.exit(), whereas Harmony does not need it.

7.1.2 Sun vs ClassPath

Comparing the Sun and ClassPath JVM libraries, we find 303 security semantic differences. Table 7.5 summarizes the results.

Sun is missing security checks in 240 cases, whereas ClassPath is missing

```
//In Harmony
...
if (DefaultCharset == null) {
    DefaultCharset = Charset.forName("ISO-8859-1");
 }
 ...
//In Sun
 ...
try {
    return encode("ISO-8859-1", ca, off, len);
} catch (UnsupportedEncodingException x) {
   System.exit(1);
   return null;
   ...
```

Figure 7.2: The portion of code present in many APIs dealing with strings in Sun and Harmony

Group	# of APIs in group	Analysis Required	Differences
1	1	Intraprocedural	Vulnerability in Harmony
2	4	Interprocedural	Vulnerability in Harmony
3	1	Intraprocedural	Vulnerability in Harmony
4	5	Interprocedural	Semantic difference
5	3	Interprocedural	Semantic difference
6	1	Interprocedural	Vulnerability in Harmony
7	1	Interprocedural	Vulnerability in Harmony
8	2	Intraprocedural	Vulnerability in Harmony

Table 7.4: Sun vs Harmony - Checks missing in Harmony

Number of Methods present both in Sun JVM and Classpath	4756
Number of Methods having security checks both in Sun JVM, and in Classpath	199
Number of Methods having security check in Sun and present in Classpath	262
Number of Methods having security check in Classpath and present in Sun	439
Number of Methods having security checks in Classpath , but no security check in Sun	240 (9 distinct groups)
Number of Methods having security check in Sun, but no security check in Classpath	63 (9 distinct gropus)
Number of differences found due to Intraprocedural Analysis	3 (2 groups)
Number of differences found due to Interprocedural Analysis	300 (16 groups)
Number of differences found due to Difference in May/Must check	0
Number of False positives reported between Sun and ClassPath	0
Number of False positives eliminated due to Interprocedural Constant Propagation	74 (4 groups)

Table 7.5: Sun vs Classpath - Static Analysis Report

security checks in 63 cases.

Sun Semantic Differences Table 7.6 depict the nature of security semantic differences found in Sun APIs in comparison to Classpath.

The Sun JVM library has 21 vulnerabilities and 219 cases are interoperability bugs. There is no false positive reported for Sun. A major chunk of APIs in ClassPath JVM libraries, load CharsetProvider class dynam-

Group	# of APIs in group	Analysis Required	Differences
1	1	Interprocedural	Vulnerability in Sun
2	1	Interprocedural	Vulnerability in Sun
3	1	Interprocedural	Semantic difference
4	9	Interprocedural	Vulnerability in Sun
5	9	Interprocedural	Vulnerability in Sun
6	5	Interprocedural	Semantic difference
7	211	Interprocedural	Semantic difference
8	1	Interprocedural	Vulnerability in Sun
9	2	Interprocedural	Semantic difference

Table 7.6: Sun vs Classpath - Checks missing in Sun

ically, whereas in Sun JVM libraries the CharsetProvider is statically loaded at the bootup time. Because of this semantic difference, ClassPath JVM libraries have extra code to perform checkPermission(new RuntimePermission ("charsetProvider")) check.

Classpath Semantic Differences Table 7.7 depicts the nature of security semantic differences found in ClassPath APIs in comparison to Sun. The ClassPath JVM library has 60 vulnerabilities and the other 3 cases are interoperability bugs. We report no false positive for the Classpath JVM library.

7.2 Limitations

Our approach has both false positives and negatives.

Group	# of APIs in group	Analysis Required	Differences
1	1	Interprocedural	Vulnerability in Classpath
2	2	Interprocedural	Vulnerability in Classpath
3	2	Interprocedural	Vulnerability in Classpath
4	1	Intraprocedural	Vulnerability in Classpath
5	3	Interprocedural	Semantic difference
6	50	Interprocedural	Vulnerability in Classpath
7	1	Interprocedural	Vulnerability in Classpath
8	2	Intraprocedural	Vulnerability in Classpath
9	1	Interprocedural	Vulnerability in Classpath

Table 7.7: Sun vs Classpath - Checks missing in Classpath

7.2.1 False Positives

In our experiment, our approach produced only 3 false positives out of 352 cases (less than 1 percent). The three false positive are due to inappropriate coding practices used in Harmony JVM libraries.

1. *java.security.Security: java.lang.String getProperty(java.lang.String)*: For returning the queried property, Sun checks for the permission using checkPermission(new SecurityPermission("getProperty."+key)) security check API. Harmony does same check using checkSecurityAccess("getProperty." + key) security check API. To check the access to system properties checkPropertyAccess() API should be used by both the implementations. This false positive results because different security APIs are used to achieve the same goal and the behavior in both implementations is exactly the same.

2. *java.net.InetAddress: boolean isReachable(java.net.NetworkInterface,int,int)*:

```
//In Harmony
public Enumeration<InetAddress> getInetAddresses() {
..
for (InetAddress element : addresses) {
            if (security != null) {
                try {
                    security.checkConnect(element.
                       getHostName(),
                            CHECK_CONNECT_NO_PORT);
                    accessibleAddresses.add(element);
                } catch (SecurityException e) {
                }
            }
        }
}
...
}
//Security check is caught and no Action taken.
//Security check inappropriately used to
//skip illegal Inet addresses.
```

Figure 7.3: Inappropriate use of security check to check accessible InetAddress

The difference in semantic behavior is due to different implementations by Harmony and Sun. Sun simply returns the result of the InetAddrImpl isReachable() API. Harmony checks reachability for different protocols by invoking isReachableByICMP(),isReachableByTCP, for example. In these methods Harmony calls getInetAddresses() to get InetAddresses. The getInetAddresses() method inappropriately uses the checkConnect() security call to validate the hostname as shown in Figure 7.3.

3. *java.net.ServerSocket: java.lang.String toString()*: In the Sun JVM li-

```
//In Harmony
public String getHostName() {
...
 SecurityManager security = System.getSecurityManager();
        try {
            if (security != null && isHostName(hostName)
               ) {
                security.checkConnect(hostName, -1);
            }
        } catch (SecurityException e) {
            return
                        Inet6Util.
                           createIPAddrStringFromByteArray

                        (ipaddress);
        }
...
}
//Security check is inappropriately used to
//choose between hostname and IPv6 Address.
```

Figure 7.4: Inappropriate use of security check to differ between hostname and IPv6 Addresses

brary toString() only returns the InetAddress, port and protocol of the ServerSocket. Harmony includes hostname in the string. The getHostName() method of InetAddress inappropriately uses a security check as shown in Figure 7.4

In general our approach can also have following false positive.

- If objects tracked by our algorithm in Section 6 are stored inside some data structure, then the entire data structure is marked tainted. Now any

operation done on this data structure is considered a security-sensitive event. This leads to reporting some extra security-sensitive events, which may not be dominated by security calls in other implementations.

7.2.2 False Negatives

Our analysis will miss the following vulnerabilities.

- Our analysis will miss security violations which occur in all implementations of the functionality. Our approach assumes at least one correct implementation of the functionality, to get useful results.

- If different arguments are passed to security check APIs in different implementations, then our approach will not be able to detect the vulnerabilities caused due to improper arguments passed to security check APIs.

- Our flow-sensitive algorithm does not report cases where the same set of security checks MAY dominate the same security-sensitive event in different implementation under different conditions. We can easily report them, but then developer intervention is required to verify whether the MAY condition for different implementation is the same or not.

- For matching security-sensitive events across different implementations, we just match the type of the data structure. If there are two implementations as shown in Figure 7.5, then our analysis will not be able report the difference in security semantics, as in both cases it will get

```
Meth1(){
 T a;
 checkWrite();
 a.data = 2;
}
and
Meth2(){
 T a;
 a.data = 2;
 checkWrite();
 T b;
 b.data=4;
}
```

Figure 7.5: A hypothetical code segment which can introduce a false negative in our approach.

the security-sensitive event “object of type T dominated by checkWrite security API”. This problem can be solved by performing slicing.

Chapter 8

Conclusion

With the use of the software as a service (SAAS) model and proprietary platforms, we see different implementations of same APIs. Our novel approach reports bugs by automatically extracting and differencing security policies of different implementations, will be increasingly applicable due to this trend. We perform interprocedural, context sensitive forward dataflow analysis to extract MAY and MUST security policies. We also perform interprocedural constant propagation and relational expression evaluation to eliminate unreachable code segments. To help the developers of the APIs, we group the security semantic bugs according to the point of error, thus providing a hint for the fix. Our approach has the following benefits:

1. We provide a promising approach of mining security policies for systems using an access control model.

2. Differencing of security policies reveals semantic security bugs, which can be either exploitable vulnerabilities or interoperability bugs.

3. Security policies extracted from different implementations can work as guidelines for future implementations.

Our experiments with different implementations of JVM libraries produced very encouraging results. We found 17 unique cases of security semantic differences (manifested in 49 APIs), between Sun and Harmony; of these 7 (manifested in 12 APIs) are vulnerabilities, 8 are interoperability bugs (manifested in 34 APIs) and 2 false positive (manifested in 3 APIs) . We have found 18 unique cases of security semantic differences (manifested in 303 APIs), between Sun and Classpath, of these 13 are vulnerabilities (manifested in 81 APIs) and 5 are interoperability bugs (manifested in 222 APIs) [36–38].

Appendices

Appendix A

Transfer Function Algorithm

We define transfer functions over each Java statement ‘v’. A statement ’v’ can contain one of the events given in Table A.1

Event	**Description**
Secure()	security check API
Native()	Native method call
Obj(member)	Non-public class member modified, i.e. assigned value or its member method invoked
Fun(member)	A method call with non-public class member as argument
Obj(argument)	Modifying a variable which is passed as argument to the root method
Fun(argument)	A method call with an argument to the root method
Return()	Return statement
Relational()	A relational operation
IF()	An IF statement
Fun()	A method call not belonging to any category above

Table A.1: Security Sensitive Events for the JVM libraries

```
If( v consist of Secure())
{
 SecureTag.api = v.method_signature;
 SecureTag.call_site = call site information;
 current_security_api_list.add(SecureTag);
```

```
}
elsif( v consist of Native())
{
  TaggedAPI.api = v.method_signature;
  TaggedAPI.security_calls =  current_security_api_list;
  tagged_api_list.add(TaggedAPI);
}
elsif( v consist of Obj(member)|Obj(argument))
{
  TaggedObject.type = Obj().type;
  TaggedObject.security_calls =  current_security_api_list;
  tagged_objects_list.add(TaggedObject);
}
elsif( v consist of Return()){
  ReturnObject.method = method_signature inside which return called;
  ReturnObject.security_calls =  current_security_api_list;
  tagged_returns.add(ReturnObject);
}
elsif( v consist of Fun(member)|Fun(argument))
{
  TaggedAPI.api = v.method_signature;
  TaggedAPI.security_calls =  current_security_api_list;
  tagged_api_list.add(TaggedAPI);
```

```
  Perform interprocedural analysis on the called function
  by passing the context of argument|member variable used.
  This is done to record events performed on
  the class members and arguments inside the called function.
}
elsif( v consist of Relational() ){
  If v consist of relational expressions like
  c = a [=, <, >, !=] b;
  where a and b are one of the following
  1. Constant Integers
  2. Constant Boolean variables
  3. Null objects
    Then our static analysis result is evaluated and accordingly
  true or false value is stored in c.
}
elsif(v consist of IF()){
    IF the condition inside IF check is boolean constant then the
  appropriate branch is choosen for the further forward flow
  analysis.
   Since the constant propogation and forward flow analysis are
  both context sensitive and interprocedural, therefore a lot of
  undesired code is rightfully excluded from the analysis. This
  help us in getting rid of reference monitor checks and
```

```
  security-sensitive events performed inside the unreachable code
  segments.
}
elsif( v is of type Fun())
{
    Perform interprocedural analysis on the called function.
}

TransferFunction(v) always return current_security_api_list at v
```

Appendix B

Detailed List of Security Related Semantic Differences

B.1 Sun vs Harmony

B.1.1 Missing checks in Sun

B.1.1.1 Group 1

Group one vulnerabilities are caused due to missing checkMemberAccess and checkPackageAccess checks in Sun. These two permissions are checked inside Class: Object newInstance() method.

```
SecurityManager sc = System.getSecurityManager();
    if (sc != null) {
        sc.checkMemberAccess(this, Member.PUBLIC);
        sc.checkPackageAccess(localReflectionData.
            packageName);
    }
```

The public/protected apis affected due to this missing check.

1. Class: Object newInstance() - The newInstance method inside Class is used to create a new instance of Class. Object is created as if operator new is used to instantiate an object with empty parameter list. All the member variables are assigned default values. The semantic difference is not a bug for Sun. This check is superfluous in Harmony. There is

already membership checks performed later using Reflection class api Reflection.checkMemberAccess(). The check dominates some of the important events like method call clone() to create copy of Class setting Reflection data like _defaultConstructor.

The check is made inside newInstance() method, therefore the missing can be detected using Normal intraprocedural forward flow analysis. But to retrieve list of the important events it dominates, interprocedural analysis is required.

B.1.1.2 Group 2

These vulnerabilities are caused due to missing checkSecurityAccess (String) called inside Security: String getProperty(String). In Harmony inside getPasswordFromCallBack() method default callback handler is loaded after reading Security.getProperty ("auth.login.defaultCallbackHandler") security property.

To access defaultCallbackHandler property the permission check is performed. In Sun the defaultCallbackHandler is not loaded dynamically. It is instantiated as a static class at the load time itself. The KeystoreSpi class defines the Service Provider Interface (SPI) for the KeyStore class. This class implemented by each cryptographic service provider who wishes to supply the implementation of a keystore for a particular keystore type.

The public/protected apis affected due to this missing check are

1. KeyStoreSpi: void engineSetEntry (String, Entry, ProtectionParameter)

2. KeyStoreSpi: void engineLoad(LoadStoreParameter)
3. KeyStoreSpi: Entry engineGetEntry (String, ProtectionParameter)
4. KeyStoreSpi: void setEntry (String, Entry, ProtectionParameter)
5. KeyStoreSpi: void load(LoadStoreParameter)
6. KeyStoreSpi: Entry getEntry (String, ProtectionParameter)

The check is made inside getPasswordFromCallBack() method of KeyStoreSpi. Therefore interprocedural dataflow analysis will be needed to capture the permission check. It is a pure semantic difference, arising due to the difference in implementations.

B.1.1.3 Group 3

The public/protected apis affected due to the checkCreateClassLoader() missing check are

1. ClassLoader: java.net.URL getResource(String)
2. ClassLoader: Enumeration getResources(String)

Some of the important events checkCreateClassLoader dominates are returning of Resource URL and getResources called on parentClassLoader. The vulnerability is caused due to the difference in the way BootstrapLoader is initialized in Sun and Harmony. The difference in behavior is due to the folllowing code

In Sun to load bootstrap classloader path following method is called

```
public static URLClassPath getBootstrapClassPath() {
AccessController.doPrivileged(
File[] classPath = getClassPath(path);
...
...
return pathToURLs(classPath);
)
}
```

In Harmony call graph end up having following method to load path for the bootstrap classloader.

```
private static void initResourceFinder() {
...
URL[] urls = new URL[urlList.size()];
resourceFinder = new URLClassLoader(urlList.toArray(urls
    ), null);
}
```

Since in Sun URLClassLoader is constructed inside the Privileged code, therefore checkCreateClassLoader() inside URLClassLoader constructor, will be executed with the JDK library permissions. The check is made inside URLClassLoader() constructor. Therefore interprocedural dataflow analysis will be needed to capture the permission check.

B.1.1.4 Group 4

The following API missing the checkPermission (new RuntimePermission (“inheritedChannel”)) check in Sun

1. System: java.nio.channels.Channel inheritedChannel()

In harmony a extra check is done before returning the inherited Channel. There is following comment by Harmony developer in the code base.
//XXX:does it mean the permission of the "access to the channel"?
//If YES then this checkPermission must be removed because it should be presented into java.nio.channels.spi.SelectorProvider.inheritedChannel()
//If NO then some other permission name (which one?) should be used here
//and the corresponding constant should be placed within
org.apache.harmony.lang.RuntimePermission class:

The check is made inside inheritedChannel(). Therefore intraprocedural dataflow analysis will be sufficient to capture this semantic difference.

B.1.1.5 Group 5

The following APIs are missing the checkPermission (ReflectPermissionCollection.SUPPRESS_ACCESS_CHECKS_PERMISSION) check.

1. ObjectOutputStream: void writeClassDescriptor(ObjectStreamClass)
2. ObjectOutputStream: void writeUnshared(Object)
3. ObjectInputStream: void defaultReadObject()
4. ObjectStreamClass: ObjectStreamClass lookup(Class)

5. ObjectOutputStream: void defaultWriteObject()
6. ObjectInputStream: GetField readFields()
7. ObjectOutputStream: void writeFields()
8. ObjectInputStream: ObjectStreamClass readClassDescriptor()
9. ObjectOutputStream: void writeObject(Object)
10. ObjectInputStream: Object readObject()
11. ObjectInputStream: Object readUnshared()
12. ObjectOutputStream: void ¡init¿(OutputStream)

The check is made inside setAccessible() method of AccessibleObject. Therefore interprocedural dataflow analysis will be needed to capture the permission check. It is a pure semantic difference, arising due to the difference in implementations.

B.1.1.6 Group 6

The following API missing checkPermission (specifyStreamHandlerPermission) check.

1. java.net.URLClassLoader: void addURL(java.net.URL)

The check is made inside URL constructor which takes streamHandler as argument. Therefore interprocedural dataflow analysis will be needed to capture

the permission check. It is a pure semantic difference, arising due to the difference in implementations. In Sun new jar search URL is not created out of provided url, but in Harmony a new search URL is created for files which are not directory or jar.

B.1.1.7 Group 7

MAY/MUST and other security check Mismatch between Harmony and Sun implementations

1. KeyStore: Builder newInstance (String,
   Provider, File,ProtectionParameter) - The API takes File as argument. In both Harmony and Sun the check is performed whether File argument passed is a file and it exists. But in Harmony inside isFile() and exists() method first the length of the path is checked. If the supplied path is empty string then false is returned. If the path has some value then the checkRead() security check is performed. The checkRead() security check is therefore optional in Harmony, i.e is done only if path of the file is non empty. The above check is not performed in Sun and hence the check is MUST for all the events happening after the call to the exists() and isFile() method. This will result in difference in semantic behavior. If the file path parameter is empty then in Sun it will result in security exception, as there will be no security policy corresponding to empty path in normal case. Whereas, in Harmony the empty path

check will return false from exists() or isFile() methods, resulting in IllegalArgumentException.

2. File: boolean mkdirs() - Same reason as above. The semantic difference will be security exception in Sun whereas in Harmony false will be returned.

3. File: String[] list(FilenameFilter) - Same reason as above. The semantic difference will be security exception in Sun whereas in Harmony null will be returned

4. File: File[] listFiles(FileFilter) - Same reason as above

5. File: File[] listFiles(FilenameFilter) - Same reason as above

The checkRead check is done inside the isFile and exists APIs. Therefore interprocedural dataflow analysis will be required to capture the permission check and important events it dominates.

B.1.1.8 Group 8

False positive

1. Security: String getProperty (String) - In Sun to check for the permission of the queried property, checkPermission security API is used. The checkPermission (new SecurityPermission(“getProperty.”+key)) is udes by Sun to check the permission for the given security property.

In Harmony the same check is performed using checkSecurityAccess (“getProperty.” + key) API call.

The checkSecurityAccess check is done inside the getProperty API itself.Therefore intraprocedural dataflow analysis will be sufficient to capture the permission check and important events it dominates. This is a False positive as the behavior in both the implementation is exactly same, just different security APIs are used to achieve the goal.

B.1.1.9 Group 9

False positive

The Sun implementation misses checkConnect in following APIs

1. java.net.InetAddress: boolean isReachable (java.net.NetworkInterface, int, int) The difference in semantic behavior is due to different implementations by Harmony and Sun. Sun just returned the result of InetAddrImpl isReachable() API.In Harmony reachability is checked for different protocols, by invoking isReachableByICMP(),isReachableByTCP etc. In these methods Harmony make call to getInetAddresses() to get InetAddresses.

```
In Harmony
public Enumeration<InetAddress> getInetAddresses()
    {
..
for (InetAddress element : addresses) {
```

```
          if (security != null) {
           try {
                  security.checkConnect(element.
                     getHostName(),
                          CHECK_CONNECT_NO_PORT);
                  accessibleAddresses.add(element);
           } catch (SecurityException e) {
           }
          }
   }
  }
   ...
   }
   \\Security check is caught and no Action taken.
   \\Security check inappropriately used to skip
   \\illegal Inet addresses.
```

2. java.net.ServerSocket: String toString() - In Sun toString() only returns the InetAddress, port and protocol of the ServerSocket. In Harmony hostname is also included in the string. In getHostName() method of InetAddress again security check is inappropriately used.

```
In Harmony
public String getHostName() {
...
 SecurityManager security = System.
    getSecurityManager();
 try {
  if (security != null && isHostName(hostName)) {
    security.checkConnect(hostName, -1);
  }
 } catch (SecurityException e) {
   return Inet6Util.createIPAddrStringFromByteArray
      (ipaddress);
 }
```

```
...
}
\\Security check is inappropriately used to
\\choose between hostname and IPv6 Address.
```

These are false positives arising due to bad coding practises in Harmony.

B.1.2 Missing checks in Harmony

B.1.2.1 Group 1

The group of following APIs missing the checkConnect in Harmony

1. java.net.URL: java.net.URLConnection openConnection(java.net.Proxy)

   - In Sun the relevant code is

   ```
   SecurityManager sm = System.getSecurityManager();
   if (proxy.type() != Proxy.Type.DIRECT && sm != null)
       {
       InetSocketAddress epoint =
       (InetSocketAddress) proxy.address();
       if (epoint.isUnresolved())
            sm.checkConnect ( epoint.getHostName(),
                      epoint.getPort());
       else
            sm.checkConnect ( epoint.getAddress().
               getHostAddress(),
       epoint.getPort());
   }
   return handler.openConnection (this, proxy);
   ```

   whereas in Harmony the code is

   return strmHandler.openConnection(this, proxy);

The check is made inside openConnection(). One of the important event is openConnection called on the private data member 'handler'. Intraprocedural dataflow analysis will be sufficient to capture this semantic difference.

B.1.2.2 Group 2

In the group of following APIs the call flow contain retrieving the HostName using getHostFromNameService (java.net.InetAddress, boolean) method. All the following methods retrieve host name either to get hash, or check equality. In Sun the check is made inside getHostFromNameService whereas in Harmony hostname is retrieved using getHostNameInternal which doesn't have checkConnect for the hostname embeded inside SocketPermission.

1. java.net.SocketPermission: boolean equals(Object)
2. java.net.SocketPermission: boolean implies(Permission)
3. java.net.SocketPermissionCollection: boolean implies(Permission)
4. java.net.SocketPermission: int hashCode()

The checkConnect is done inside the getHostFromNameService() API.Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.1.2.3 Group 3

The following API miss the checkAccept check before accepting the connection on the provided socket.

1. java.net.ServerSocket: void implAccept(java.net.Socket) - In Sun the code is

   ```
   si = s.impl;
   s.impl = null;
   si.address = new InetAddress();
   si.fd = new FileDescriptor();
   getImpl().accept(si);
   SecurityManager security = System.getSecurityManager
       ();
   if (security != null) {
        security.checkAccept(si.getInetAddress().
           getHostAddress(),
         si.getPort());
   }
     ..
   s.impl = si;
   s.postAccept();
   ```

   In Harmony the code is

   ```
   impl.accept(aSocket.impl);
   aSocket.accepted();
   ```

The check is made inside implAccept(). One of the important event is seting impl field of the parameter 's' and calling methods like postAccept and accept on private data member. Intraprocedural dataflow analysis will be sufficient to capture this semantic difference.

B.1.2.4 Group 4

The following APIs miss the checkExit check in Harmony JVM.

1. String: void init(byte[])
2. String: byte[] getBytes()
3. String: void init(byte[],int,int)
4. MessageDigest: String toString()
5. ByteArrayOutputStream: String toString()

This is interesting semantic difference between Harmony and Sun which arises, due to the difference in the way they react to an Exception. In Sun if the default "ISO-8859-1" charset decoder is not present then Application is terminated by calling System.exit(). Whereas in Harmony the UnsupportedEncodingException is thrown. To perform System.exit() the user require Exit permission, and hence Sun has checkExit check inside System.exit(), whereas Harmony does not need it.

The checkExit is done inside the System.exit() API.Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.1.2.5 Group 5

The following group of APIs are missing checkConnect check which is made as part of retrieving hostname from the given InetAddress.

1. java.net.URLStreamHandler: int hashCode(java.net.URL)
2. cert.X509Certificate: Collection getSubjectAlternativeNames()
3. cert.X509Certificate: Collection getIssuerAlternativeNames()

The above APIs are generating a string out of the given URLs. In Sun if hostname is present then these APIs try to resolve the hostname into IPAddress, and resolved IPAddress is made part of the final String. In Harmony the hostname is taken as it is. Due to this difference in semantic behavior, the check is present in Sun, while creating InetAddress out of the given URL, but Harmony doesn't have equivalent code.

The checkConnect is done inside the InetAddress.getByName() API. Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.1.2.6 Group 6

The following API missing the checkSecurityAccess ("putProviderProperty." + name) check

1. Provider: void load(InputStream) - API is used for reading the Property list from the given InputStream.

The checkSecurityAccess is done inside the check API.Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.1.2.7 Group 7

The following API missing the checkPermission (SecurityConstants. GET_POLICY_PERMISSION) check.

1. ProtectionDomain: String toString() - The ProtectionDomain class encapsulates the characteristics of a domain, which encloses a set of classes whose instances are granted a set of permissions when being executed on behalf of a given set of Principals. toString() API can give out Permissions associated with the ProtectionDomain, therefore the check for retrieving security permissions is required.

The check is done inside the seeAllp API.Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.1.2.8 Group 8

The following API missing checkAccept check in Harmony, whereas check is present in Sun.

1. java.net.DatagramSocket: void connect(java.net.SocketAddress) - Both in Sun and Harmony, connect() has checkListen(),checkConnect and checkMulticast check. In Sun before establishing connection checkAccept check is also done for the other InetAddress

2. java.net.DatagramSocket: void connect(java.net.InetAddress,int) - same as above

Intraprocedural dataflow analysis will be sufficient to capture the missing permission check .

B.1.3 False positives eliminated due to interprocedural constant propagation and relational expression folding in Sun

B.1.3.1 Group 1

The following APIs call checkClosedAndBind(boolean bind) with bind as false. If bind is true then this API try to check whether connect is bound or not. If not bound then connection is made, and hence the checkListen() check is performed.

1. java.net.DatagramSocket: void setReceiveBufferSize(int)
2. java.net.DatagramSocket: boolean getBroadcast()
3. java.net.DatagramSocket: int getReceiveBufferSize()
4. java.net.DatagramSocket: void setReuseAddress(boolean)
5. java.net.DatagramSocket: int getTrafficClass()
6. java.net.DatagramSocket: void setSendBufferSize(int)
7. java.net.DatagramSocket: void setTrafficClass(int)
8. java.net.DatagramSocket: void setBroadcast(boolean)
9. java.net.DatagramSocket: int getSoTimeout()
10. java.net.DatagramSocket: int getSendBufferSize()

11. java.net.DatagramSocket: boolean getReuseAddress()

12. java.net.DatagramSocket: void setSoTimeout(int)

B.1.4 False positives eliminated due to interprocedural constant propagation and relational expression folding in Harmony

B.1.4.1 Group 1

The following APIs call constructor URL (String protocol, String host, int port, String file, URLStreamHandler handler) with handler as null. If handler is not null then
checkPermission (SecurityConstants. SPECIFY_HANDLER_PERMISSION) check is performed for handler.

1. Signature: Signature getInstance
   (String, Provider)

2. Signature: Signature getInstance (String, String)

B.1.4.2 Group 2

The following APIs call constructor URL (URL context, String spec, URLStreamHandler handler) with handler as null. If handler is not null then checkPermission (SecurityConstants. SPECIFY_HANDLER_PERMISSION) check is performed for handler.

1. javax.xml.parsers.DocumentBuilderFactory: javax.xml.validation.Schema getSchema()

2. javax.xml.parsers.DocumentBuilder: boolean isXIncludeAware()
3. Package: boolean isSealed()
4. Package: String toString()
5. Package: String getSpecificationVendor()
6. Package: boolean isSealed(java.net.URL)
7. Package: String getImplementationTitle()
8. Package: String getImplementationVendor()
9. Package: boolean isCompatibleWith(String)
10. Package: String getSpecificationVersion()
11. javax.xml.parsers.DocumentBuilder: void reset()
12. Package: String getImplementationVersion()
13. Package: String getSpecificationTitle()
14. javax.xml.parsers.DocumentBuilder: javax.xml.validation.Schema getSchema()
15. javax.xml.parsers.DocumentBuilderFactory:
    void setSchema (javax.xml.validation.Schema)
16. javax.xml.parsers.DocumentBuilderFactory: void setXIncludeAware(boolean)

17. javax.xml.parsers.DocumentBuilderFactory: boolean isXIncludeAware()

B.2 Sun vs Classpath

B.2.1 Missing checks in Sun

B.2.1.1 Group 1

The following API misses the checkSecurityAccess (keystore.type) check.

1. KeyStore: String getDefaultType() - The APi is used to returns the default keystore type as specified in the Java security properties file, or the string. The default keystore type can be changed by setting the value of the "keystore.type" security property (in the Java security properties file) to the desired keystore type.

   The Security.getProperty is called in Sun, inside the Privileged code section.

```
public final static String getDefaultType() {
    String kstype;
    kstype = AccessController.doPrivileged (
        new PrivilegedActionString() {
    public String run() {
      return Security.getProperty(KEYSTORE_TYPE);
    }
    });
    if (kstype == null) {
        kstype = "jks";
    }
    return kstype;
}
```

Whereas in Classpath JVM the Security.getProperty is not inside the Privileged section.

```
public static final String getDefaultType()
  {
    String tmp = Security.getProperty("keystore.type
       ");

    if (tmp == null)
      tmp = "gkr";
    return tmp;
  }
```

Therefore the code will run with permission of JVM in Sun, whereas in Classpath the code will run with the permissions of the calling Protection Domain.

The check is done inside the Security.getProperty() API. Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.2 Group 2

The following API is missing the checkPermission(new LoggingPermission("control",null))

1. logging.LogManager: boolean addLogger (logging.Logger) - The addLogger method call doSetParent to set the parent. The operation is performed under Privileged block. It is not clear why unprivileged user

should be allowed to set logger of its choice. Malicious user can set any logger of his choice to retrieve the sensitive log information.

In Sun the code is

```
public synchronized boolean addLogger(Logger logger)
    {
...
 if (parent != null) {
    doSetParent(logger, parent);
 }
...
}

private static void doSetParent (final Logger logger
   ,
 final Logger parent) {
...
 AccessController.doPrivileged(new
    PrivilegedActionObject() {
 public Object run() {
    logger.setParent(parent);
    return null;
 }});
}
```

In Classpath equivalent code is

```
public synchronized boolean addLogger(Logger logger)
   {
...
  if ((name != null) && ! name.equals(""))
      checkAccess();

    Logger parent = findAncestor(logger);
    loggers.put(name, new WeakReferenceLogger(logger
       ));
    if (parent != logger.getParent())
```

```
        logger.setParent(parent);
    ...
}
```

The check is doen inside checkAccess() API

The check is done inside the checkAccess() API.Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.3 Group 3

The following API is missing the checkConnect() check

1. CodeSource: boolean implies (CodeSource) - Returns true if this CodeSource object "implies" the specified CodeSource.

Its a difference in implementation. SocketPermission class gets name of port provided via InetAddress. There are different approach to match certificates in both of the JVMs. In classpath each field is matched whereas in Sun byte stream is matched.

The check is done inside the InetAddress.getAllByName(). Therefore interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.4 Group 4

The following APIs missing the checkRead() check performed as part of File operations

1. TimeZone: String[] getAvailableIDs() - readZoneInfoFile is read inside Privileged code in Sun.
2. TimeZone: String[] getAvailableIDs(int)
3. TimeZone: TimeZone getTimeZone(String)
4. ClassLoader: java.net.URL getResource (String) - In Sun getBootstrap-ClassPath() method construct URLs inside Privileged code.
5. ClassLoader: Enumeration getResources (String)
6. regex.Pattern: String[] split (CharSequence,int)
7. java.net.URLClassLoader: java.net.URLClassLoader newInstance (java.net.URL[], ClassLoader) - Sun doesn't make check while making new Instance of the classloader. It assumes anyway we will have check-PackageAccess at the time of loading classes.
8. java.net.URLClassLoader: java.net.URLClassLoader newInstance(java.net.URL[])
9. java.net.URLClassLoader: void addURL (java.net.URL) - newInstance called inside addURL.

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.5 Group 5

The following APIs missing checkPermission (new RuntimePermission(“selectorProvider”)) check

1. java.nio.channels.spi.SelectorProvider: java.nio.channels.Channel inheritedChannel()
2. System: java.nio.channels.Channel inheritedChannel()
3. java.nio.channels.SocketChannel: java.nio.channels.SocketChannel open (java.net.SocketAddress)
4. java.nio.channels.DatagramChannel: java.nio.channels.DatagramChannel open()
5. java.nio.channels.Selector: java.nio.channels.Selector open()
6. java.nio.channels.Pipe: java.nio.channels.Pipe open()
7. java.nio.channels.spi.SelectorProvider: java.nio.channels.spi.SelectorProvider provider()
8. java.nio.channels.SocketChannel: java.nio.channels.SocketChannel open()
9. java.nio.channels.ServerSocketChannel: java.nio.channels.ServerSocketChannel open()

In Sun loadProviderFromProperty is performed inside privileged code.

```
public static SelectorProvider provider() {
synchronized (lock) {
if (provider != null)
    return provider;
    return (SelectorProvider)AccessController
    .doPrivileged(new PrivilegedAction() {
    public Object run() {
    if (loadProviderFromProperty())
        return provider;
    if (loadProviderAsService())
        return provider;
    provider = sun.nio.ch.DefaultSelectorProvider.create
       ();
    return provider;
    }
});
}
}
```

whereas in Classpath

```
public static synchronized SelectorProvider provider()
{
...
 String propertyValue =
 System.getProperty ( "java.nio.channels.spi.
    SelectorProvider" );
...
}
```

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.6 Group 6

The following APIs are missing checkRead() check as part of FileInputStream: void init(File) constructor.

1. javax.xml.parsers.DocumentBuilder: org.w3c.dom.Document parse (File) - The difference in semantics is due to the difference in implementation. In Classpth FileInputStream is created out of the File argument provided, whereas in Sun a URL is constructed out of File and then data is loaded.

2. javax.xml.parsers.DocumentBuilderFactory: javax.xml.parsers.DocumentBuilderFactory newInstance() - Use FileInputstream to read file pointed by " javax.xml.parsers.DocumentBuilderFactory " property dynamically.

3. Properties: void loadFromXML(InputStream) - Uses DocumentBuilderFactory.newInstance()

4. java.net.URLConnection: java.net.FileNameMap getFileNameMap() - In Sun the FileNameMap table is statically provided using sun.net.www.MimeTable.loadTable() api. In ClassPath the FileNameMap is loaded at runtime by reading SystemProperties.getProperty ("gnu.classpath.mime.types.file") value.

5. java.net.URLConnection: String guessContentTypeFromName (String) - Use getFileNameMap.

All of the above are semantic differences caused due to some difference in implementation. Mostly due to dynamic upload of some file in Classpath, which is done statically in Sun.

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.7 Group 7

The following APIs are missing checkPermission (new RuntimePermission (“charsetProvider”)) as part of java.nio.charset.spi.CharsetProvider: void init() constructor.

1. jar.JarFile: InputStream getInputStream (zip.ZipEntry)
2. java.text.DateFormat: java.text.DateFormat getDateTimeInstance()
3. ObjectInputStream: int read()
4. ObjectOutputStream: void writeClassDescriptor (ObjectStreamClass)
5. java.text.SimpleDateFormat: void init (String)
6. PrintStream: void print (String)
7. logging.Handler: void setFilter (logging.Filter)
8. PrintStream: void println (int)

9. logging.Logger: void finer (String)
10. java.text.SimpleDateFormat: StringBuffer format (Date, StringBuffer, java.text.FieldPosition)
11. ObjectOutputStream: void writeUnshared (Object)
12. jar.Manifest: void init (InputStream)
13. regex.Matcher: String replaceFirst (String)
14. regex.Matcher: String replaceAll (String)
15. logging.Logger: void entering (String, String)
16. logging.Logger: void log (logging.Level, String, Object[])
17. java.text.MessageFormat: void applyPattern (String)
18. logging.Handler: void setErrorManager (logging.ErrorManager)
19. java.text.DateFormat: java.text.DateFormat getDateInstance()
20. logging.Logger: void info(String)
21. logging.Logger: void log (logging.Level, String, Object)
22. java.text.DateFormat: java.text.DateFormat getTimeInstance (int, Locale)
23. PrintStream: void print(double)

24. java.net.URI: java.net.URI create(String)
25. java.text.DateFormat: java.text.DateFormat getTimeInstance()
26. String: boolean matches(String)
27. PrintWriter: void init(OutputStream)
28. logging.Logger: void logrb (logging.Level, String, String, String, String, Object)
29. java.net.URI: void init (String, String, String, String)
30. PrintStream: void println(long)
31. Calendar: String getDisplayName (int, int, Locale)
32. PrintStream: void print(char)
33. Date: String toGMTString()
34. logging.Handler: void setFormatter (logging.Formatter)
35. logging.Logger: void severe(String)
36. java.net.URI: java.net.URI normalize()
37. logging.Logger: void fine(String)
38. java.net.JarURLConnection: jar.JarEntry getJarEntry()
39. PrintStream: void println(boolean)

40. jar.Manifest: void read(InputStream)
41. ServiceLoader: boolean hasNext()
42. java.net.URI: java.net.URI resolve(String)
43. java.text.DateFormatSymbols: java.text.DateFormatSymbols getInstance (Locale)
44. OutputStreamWriter: void init(OutputStream,String)
45. java.net.JarURLConnection: cert.Certificate[] getCertificates()
46. cert.PolicyQualifierInfo: void init(byte[])
47. logging.Logger: void setFilter(logging.Filter)
48. java.net.URI: java.net.URI parseServerAuthority()
49. logging.Formatter: String formatMessage (logging.LogRecord)
50. PrintStream: void print(int)
51. regex.Matcher: boolean matches()
52. java.net.URLConnection: long getHeaderFieldDate (String, long)
53. logging.Logger: void setParent(logging.Logger)
54. java.net.URI: void init (String, String, String)

55. logging.Logger: void finest(String)

56. PrintStream: void println(double)

57. logging.Logger: void logrb (logging.Level, String, String, String, String, Object[])

58. String: String format (Locale, String, Object[])

59. logging.Logger: void config(String)

60. Date: String toString()

61. ThreadGroup: void uncaughtException (Thread, Throwable)

62. ObjectInputStream: ObjectStreamClass readClassDescriptor()

63. ObjectInputStream: int available()

64. TimeZone: String getDisplayName(Locale)

65. java.net.URL: java.net.URI toURI()

66. java.net.URLConnection: long getLastModified()

67. jar.Manifest: void write(OutputStream)

68. javax.security.auth.x500.X500Principal: byte[] getEncoded()

69. logging.Logger: void warning(String)

70. String: String format (String, Object[])

71. jar.JarFile: jar.JarEntry getJarEntry(String)

72. ObjectInputStream: int read (byte[], int, int)

73. regex.Pattern: boolean matches(String, CharSequence)

74. String: String replaceFirst (String, String)

75. java.text.MessageFormat: void init (String, Locale)

76. PrintStream: void print(char[])

77. java.nio.charset.Charset: java.nio.charset.Charset forName(String)

78. java.net.URI: void init (String, String, String, int, String, String, String)

79. java.net.URLConnection: long getExpiration()

80. java.text.MessageFormat: String format (String, Object[])

81. ObjectOutputStream: void writeObject(Object)

82. logging.Logger: void exiting(String,String,Object)

83. java.net.HttpURLConnection: long getHeaderFieldDate(String,long)

84. String: String[] split(String)

85. logging.Logger: void entering (String, String, Object[])

86. java.text.DateFormatSymbols: void init()

87. logging.LogManager: logging.LogManager getLogManager()

88. ServiceLoader: Object next()

89. java.net.JarURLConnection: jar.Manifest getManifest()

90. PrintStream: void print(float)

91. Properties: void load(InputStream)

92. logging.Logger: void logp (logging.Level, String, String, String)

93. java.text.MessageFormat: StringBuffer format (Object, StringBuffer, java.text.FieldPosition)

94. java.text.DateFormat: java.text.DateFormat getDateTimeInstance(int,int)

95. logging.Logger: void logrb (logging.Level, String, String, String, String, Throwable)

96. ObjectInputStream: Object readUnshared()

97. Properties: void storeToXML(OutputStream,String)

98. java.text.MessageFormat: Object[]
parse (String, java.text.ParsePosition)

99. PrintStream: void println()

100. PrintStream: PrintStream printf (Locale, String, Object[])

101. regex.Matcher: regex.Matcher reset(CharSequence)

102. logging.Logger: logging.Logger
getLogger (String,String)

103. java.text.SimpleDateFormat: java.text.AttributedCharacterIterator
formatToCharacterIterator (Object)

104. java.net.JarURLConnection: jar.Attributes getAttributes()

105. PrintStream: void print(Object)

106. logging.Logger: void exiting(String,String)

107. logging.Logger: void logrb (logging.Level, String, String, String, String
)

108. PrintStream: PrintStream printf (String, Object[])

109. OutputStreamWriter: void init(OutputStream)

110. PrintStream: void println(char)

111. java.text.MessageFormat: Object[] parse(String)

112. ObjectInputStream: void defaultReadObject()

113. Formatter: void init(OutputStream)

114. TimeZone: String getDisplayName (boolean, int, Locale)

115. logging.ErrorManager: void error(String, Exception, int)

116. jar.JarFile: zip.ZipEntry getEntry(String)
117. ByteArrayOutputStream: String toString (String)
118. regex.Pattern: regex.Pattern compile (String, int)
119. java.nio.charset.Charset: boolean isSupported(String)
120. java.net.URLConnection: long getDate()
121. PrintStream: PrintStream append (CharSequence, int, int)
122. java.net.URI: java.net.URI relativize(java.net.URI)
123. Thread: void dumpStack()
124. TimeZone: String getDisplayName (boolean, int)
125. PrintStream: PrintStream append (CharSequence)
126. regex.Matcher: boolean find()
127. java.text.DateFormat: java.text.DateFormat getDateTimeInstance (int, int, Locale)
128. javax.security.auth.x500.X500Principal: void init(InputStream)
129. java.text.DateFormatSymbols: void init(Locale)
130. regex.Pattern: regex.Pattern compile(String)
131. logging.Logger: void log(logging.LogRecord)

132. java.text.SimpleDateFormat: Date
parse (String, java.text.ParsePosition)
133. PrintStream: Appendable append(char)
134. java.nio.charset.Charset: SortedMap availableCharsets()
135. java.net.URI: void init (String, String, String, String, String)
136. PrintStream: void print(boolean)
137. PrintStream: PrintStream format (Locale, String, Object[])
138. String: void init (byte[], int, int, String)
139. java.net.JarURLConnection: jar.Attributes getMainAttributes()
140. Properties: void storeToXML (OutputStream, String, String)
141. logging.Logger: void setLevel(logging.Level)
142. cert.TrustAnchor: void init (String, PublicKey, byte[])
143. PrintStream: void println(float)
144. PrintStream: void println(Object)
145. Properties: void save (OutputStream, String)
146. ObjectOutputStream: void defaultWriteObject()
147. java.text.MessageFormat: void init(String)

148. regex.Pattern: regex.Matcher matcher (CharSequence)

149. java.text.DateFormat: java.text.DateFormat getInstance()

150. regex.Matcher: boolean lookingAt()

151. ObjectInputStream: GetField readFields()

152. logging.Handler: void setLevel(logging.Level)

153. javax.security.auth.x500.X500Principal: void init(byte[])

154. ThreadGroup: void list()

155. logging.Logger: void addHandler(logging.Handler)

156. Throwable: void printStackTrace()

157. java.text.MessageFormat: Object
     parseObject (String, java.text.ParsePosition)

158. PrintWriter: PrintWriter format (Locale, String, Object[])

159. PrintStream: Appendable append (CharSequence, int, int)

160. logging.Logger: void entering (String, String, Object)

161. PrintStream: PrintStream format (String, Object[])

162. Formatter: void init (OutputStream, String)

163. logging.Logger: void setUseParentHandlers(boolean)

164. java.text.DateFormat: java.text.DateFormat getDateInstance (int, Locale)

165. java.net.URI: java.net.URI resolve(java.net.URI)

166. logging.Logger: void logp (logging.Level, String, String, String, Object)

167. PrintStream: void print(long)

168. java.net.URI: void init(String)

169. logging.Logger: void logp (logging.Level, String, String, String, Throwable)

170. logging.Logger: void removeHandler(logging.Handler)

171. regex.Pattern: String[] split(CharSequence)

172. PrintStream: PrintStream append(char)

173. logging.Handler: void setEncoding(String)

174. jar.JarFile: jar.Manifest getManifest()

175. String: String[] split (String, int)

176. TimeZone: String getDisplayName()

177. InputStreamReader: void init (InputStream, String)

178. ObjectInputStream: void readStreamHeader()

179. logging.Handler: logging.ErrorManager getErrorManager()

180. java.text.DateFormat: java.text.DateFormat getTimeInstance(int)

181. java.text.MessageFormat: java.text.AttributedCharacterIterator format-ToCharacterIterator (Object)

182. Date: String toLocaleString()

183. java.net.URI: String toASCIIString()

184. PrintStream: void println(char[])

185. PrintStream: void init (OutputStream, boolean, String)

186. String: String replaceAll (String, String)

187. PrintStream: Appendable append(CharSequence)

188. logging.Logger: logging.Logger getLogger(String)

189. String: byte[] getBytes(String)

190. Properties: void store(OutputStream,String)

191. java.text.DateFormat: java.text.DateFormat getDateInstance(int)

192. java.nio.charset.Charset: java.nio.charset.Charset defaultCharset()

193. javax.security.auth.x500.X500Principal: void init(String)

194. ObjectInputStream: Object readObject()

195. InputStreamReader: void init(InputStream)
196. logging.Logger: void log (logging.Level, String, Throwable)
197. PrintStream: void println(String)
198. logging.Logger: void logp (logging.Level, String, String, String, Object[])
199. logging.Logger: void log (logging.Level, String)
200. Formatter: Formatter format (Locale, String, Object[])
201. Calendar: Map getDisplayNames (int, int, Locale)
202. java.text.MessageFormat: StringBuffer format (Object[], StringBuffer, java.text.FieldPosition)
203. java.text.SimpleDateFormat: void init (String, Locale)
204. regex.Matcher: boolean find(int)
205. java.text.SimpleDateFormat: void init()
206. java.text.DateFormatSymbols: java.text.DateFormatSymbols getInstance()
207. Formatter: void init (OutputStream, String, Locale)
208. String: void init(byte[],String)
209. java.text.MessageFormat: void setLocale(Locale)
210. Formatter: Formatter format (String, Object[])

211. logging.Logger: void throwing (String, String, Throwable)

All of the above APIs use CharsetProvider to decode some string. In Classpath CharsetProvider is loaded dynamically and hence there is check for loading custom Charsetprovider. In Sun there is no feature for dynamiclly uploading CharsetProvider.

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates.

B.2.1.8 Group 8

The following API missing checkRead check in Sun, whereas check is present in Classpath.

1. Runtime: void loadLibrary (String) - Both in Sun and Classpath, loadLibrary has checkLink() check. In Classpath before loading the library a private loadLib function is called, which perform checkRead() check before calling native method to load the library.

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check is performed inside loadLib() method.

B.2.1.9 Group 9

The following API missing checkConnect() check in Sun, whereas check is present in Classpath.

1. java.net.Socket: java.net.InetAddress getLocalAddress() - In getLocalAddress checkConnect security check is performed in classpath.

2. java.net.Socket: java.net.SocketAddress getLocalSocketAddress() - getLocalAddress method is called from getLocalSocketAddress ,and hence the checkConnect is present interprocedurally.

Interprocedural dataflow analysis will be required for getLocalSocketAddress, whereas Intraprocedural analysis will be sufficient for getLocalAddress.
The check may be unwanted. In Classpath code base we found following comment above the check.
// FIXME: According to libgcj, checkConnect() is supposed to be called
// before performing this operation. Problems: 1) We don't have the
// addr until after we do it, so we do a post check. 2). The docs I
// see don't require this in the Socket case, only DatagramSocket, but
// we'll assume they mean both.

B.2.2 Missing checks in Classpath

B.2.2.1 Group 1

In the follwoign API constructor missed the sm.checkPermission (SUBCLASS_IMPLEMENTATION_PERMISSION) check.

1. ObjectOutputStream: void init(OutputStream) - The outputStream constructor miss the check for subclass implementation permission.

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check in performed inside ObjectOutputStream: void verifySubclass() method.

B.2.2.2 Group 2

The following APIs missing the checkPermission(new LoggingPermission("control"))

1. logging.LogManager: void addPropertyChangeListener (java.beans.PropertyChangeListener)
2. logging.LogManager: void removePropertyChangeListener (java.beans.PropertyChangeListener)

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check in performed inside checkAccess() method of LogManager.

B.2.2.3 Group 3

The following APIs missing the checkConnect() check

1. java.net.SocketPermission: boolean equals(Object)
2. java.net.SocketPermission: int hashCode()

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check in performed inside checkAccess() method of LogManager.

B.2.2.4 Group 4

The following API is missing checkRead() check

1. File: boolean isHidden()

Intraprocedural dataflow analysis will be sufficient to capture the permission check as the check in performed inside isHidden() method.

B.2.2.5 Group 5

The following APIs missing checkExit() check

1. cert.PolicyQualifierInfo: String toString()
2. MessageDigest: String toString()
3. jar.JarFile: InputStream getInputStream(zip.ZipEntry)

The difference is due to the way in which JVMs react to an Exception. In Sun if the default "ISO-8859-1" charset decoder is not present then Application is terminated by calling System.exit(). Whereas in Classpath the UnsupportedEncodingException is thrown. To perform System.exit() the user require Exit permission, and hence Sun has checkExit check inside System.exit(), whereas Classpath does not need it.
Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check is performed inside System.exit().

B.2.2.6 Group 6

The following APIs missing the checkConnect() check

1. javax.crypto.Cipher: byte[] doFinal(byte[],int,int)
2. javax.crypto.ExemptionMechanism: javax.crypto.ExemptionMechanism getInstance(String,String)
3. javax.crypto.Cipher: int doFinal(byte[],int,int,byte[])
4. javax.crypto.Cipher: Provider getProvider()
5. java.net.SocketPermission: boolean equals(Object)
6. javax.crypto.Cipher: byte[] update(byte[],int,int)
7. java.net.URL: boolean equals(Object)

8. javax.crypto.Cipher: byte[] update(byte[])
9. javax.crypto.Cipher: AlgorithmParameters getParameters()
10. javax.crypto.Cipher: byte[] doFinal(byte[])
11. javax.crypto.Cipher: void init(int,Key)
12. javax.crypto.Cipher: int doFinal(byte[],int)
13. java.net.URLStreamHandler: boolean sameFile(java.net.URL,java.net.URL)
14. javax.crypto.ExemptionMechanism: javax.crypto.ExemptionMechanism getInstance (String, Provider)
15. javax.crypto.Cipher: byte[] getIV()
16. Package: boolean isSealed(java.net.URL)
17. javax.crypto.Cipher: void init (int, Key, spec.AlgorithmParameterSpec)
18. javax.crypto.Cipher: void init (int,Key,SecureRandom)
19. javax.crypto.Cipher: void init (javax.crypto.CipherSpi, Provider, String)
20. javax.crypto.Cipher: int update (byte[], int, int, byte[], int)

21. javax.crypto.Cipher: javax.crypto.Cipher
    getInstance(String)

22. java.net.URLStreamHandler: boolean equals (java.net.URL,java.net.URL)

23. javax.crypto.Cipher: int update (java.nio.ByteBuffer, java.nio.ByteBuffer)

24. javax.crypto.Cipher: void
    init (int, cert.Certificate, SecureRandom)

25. java.net.URL: int hashCode()

26. java.net.URLStreamHandler: int hashCode(java.net.URL)

27. javax.crypto.Cipher: Key unwrap (byte[], String,int)

28. javax.crypto.Cipher: void
    init (int, Key, spec.AlgorithmParameterSpec,
    SecureRandom)

29. javax.crypto.Cipher: int update(byte[],int,int,byte[])

30. javax.crypto.Cipher: byte[] wrap(Key)

31. java.net.SocketPermission: int hashCode()

32. CodeSource: boolean equals(Object)

33. javax.crypto.Cipher: void
    init (int, Key, AlgorithmParameters)

34. javax.crypto.Cipher: int doFinal(java.nio.ByteBuffer,java.nio.ByteBuffer)
35. javax.crypto.Cipher: javax.crypto.ExemptionMechanism getExemption-Mechanism()
36. javax.crypto.Cipher: byte[] doFinal()
37. javax.crypto.Cipher: int doFinal(byte[],int,int,byte[],int)
38. cert.X509Certificate: Collection getSubjectAlternativeNames()
39. javax.crypto.ExemptionMechanism: javax.crypto.ExemptionMechanism getInstance (String)
40. javax.crypto.Cipher: javax.crypto.Cipher getInstance (String, Provider)
41. CodeSource: int hashCode()
42. cert.X509Certificate: Collection getIssuerAlternativeNames()
43. javax.crypto.Cipher: int getBlockSize()
44. javax.crypto.Cipher: int getOutputSize(int)
45. javax.crypto.Cipher: void
    init (int, Key, AlgorithmParameters,
    SecureRandom)
46. Signature: Signature getInstance (String, Provider)
47. java.net.URL: boolean sameFile(java.net.URL)

48. javax.crypto.Cipher: javax.crypto.Cipher getInstance (String, String)

49. javax.crypto.Cipher: void init (int, cert.Certificate)

50. Signature: Signature getInstance (String, String)

The above apis instead of calling java.net.InetAddress[] getAllByName0() to retrieve hostnames, directly access the hostname field in respective classes.

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check is performed inside getAllByName0() method.

B.2.2.7 Group 7

The following APIs missing the checkPermission (SecurityConstants. GET_POLICY_PERMISSION) for dynamically loading policy permission.

1. ProtectionDomain: String toString()

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check is performed inside seeAllp()

B.2.2.8 Group 8

The following APIs missing the checkConnect()

1. java.net.Socket: void connect(java.net.SocketAddress)

2. java.net.Socket: void connect(java.net.SocketAddress,int)

Intraprocedural dataflow analysis will be sufficient to capture the permission check as the check in performed inside connect methods.

B.2.2.9 Group 9

The following API missing checkMulticast,checkAccept and checkListen check in Classpath, whereas check is present in Sun.

1. java.net.DatagramSocket: void connect(java.net.InetAddress,int) - Both in Sun and Classpath, connect() has checkConnect() check. In Sun before establishing connection checkMulticast is done for multicast addresses and, checkAccept and checkListen done for the other InetAddress

Interprocedural dataflow analysis will be needed to capture the permission check and important events it dominates, as the check is performed inside connectInternal() method.

B.2.3 False positives eliminated due to interprocedural constant propagation and relational expression folding in Sun

B.2.3.1 Group 1

The following APIs call checkClosedAndBind(boolean bind) with bind as false. If bind is true then this API try to check whether connect is bound or not. If not bound then connection is made, and hence the checkListen() check is performed.

1. java.net.DatagramSocket: void setReceiveBufferSize(int)
2. java.net.DatagramSocket: boolean getBroadcast()
3. java.net.DatagramSocket: int getReceiveBufferSize()
4. java.net.DatagramSocket: void setReuseAddress(boolean)
5. java.net.DatagramSocket: int getTrafficClass()
6. java.net.DatagramSocket: void setSendBufferSize(int)
7. java.net.DatagramSocket: void setTrafficClass(int)
8. java.net.DatagramSocket: void setBroadcast(boolean)
9. java.net.DatagramSocket: int getSoTimeout()
10. java.net.DatagramSocket: int getSendBufferSize()
11. java.net.DatagramSocket: boolean getReuseAddress()
12. java.net.DatagramSocket: void setSoTimeout(int)

B.2.4 False positives eliminated due to interprocedural constant propagation and relational expression folding in Classpath

B.2.4.1 Group 1

The following APIs call getAllByName0 (String host, InetAddress reqAddr, boolean check) with check as false. If check is true then checkConnect check is performed.

1. cert.PKIXParameters: void init(KeyStore)

2. java.net.SocketPermission: boolean equals(Object)

3. java.net.URLStreamHandler: boolean sameFile(java.net.URL,java.net.URL)

4. java.net.URLStreamHandler: boolean equals(java.net.URL,java.net.URL)

5. java.net.URLStreamHandler: int hashCode(java.net.URL)

6. java.net.SocketPermission: int hashCode()

7. cert.X509Certificate: Collection getSubjectAlternativeNames()

8. cert.TrustAnchor: void init(cert.X509Certificate,byte[])

9. cert.X509Certificate: Collection getIssuerAlternativeNames()

B.2.4.2 Group 2

The following APIs call constructor URL(String protocol, String host, int port, String file, URLStreamHandler handler) with handler as null. If handler is not null then
checkPermission (SecurityConstants. SPECIFY_HANDLER_PERMISSION) check is performed for handler.

1. javax.crypto.Cipher: byte[] doFinal(byte[],int,int)

2. javax.crypto.ExemptionMechanism: javax.crypto.ExemptionMechanism getInstance (String, String)

3. javax.crypto.Cipher: int doFinal(byte[],int,int,byte[])
4. javax.crypto.Cipher: Provider getProvider()
5. javax.crypto.Cipher: byte[] update(byte[],int,int)
6. javax.crypto.Cipher: byte[] update(byte[])
7. javax.crypto.Cipher: AlgorithmParameters getParameters()
8. javax.crypto.Cipher: byte[] doFinal(byte[])
9. javax.crypto.Cipher: void init(int,Key)
10. javax.crypto.Cipher: int doFinal(byte[],int)
11. javax.crypto.ExemptionMechanism: javax.crypto.ExemptionMechanism getInstance (String, Provider)
12. javax.crypto.Cipher: byte[] getIV()
13. javax.crypto.Cipher: void init (int, Key, spec.AlgorithmParameterSpec)
14. javax.crypto.Cipher: void init (int, Key, SecureRandom)
15. javax.crypto.Cipher: int update (byte[], int, int, byte[], int)
16. javax.crypto.Cipher: javax.crypto.Cipher getInstance (String)
17. javax.crypto.Cipher: int update (java.nio.ByteBuffer, java.nio.ByteBuffer)

18. javax.crypto.Cipher: void init (int, cert.Certificate, SecureRandom)

19. javax.crypto.Cipher: Key unwrap (byte[], String, int)

20. javax.crypto.Cipher: void init (int, Key, spec.AlgorithmParameterSpec, SecureRandom)

21. javax.crypto.Cipher: int update(byte[],int,int,byte[])

22. javax.crypto.Cipher: byte[] wrap(Key)

23. javax.crypto.Cipher: void init (int, Key, AlgorithmParameters)

24. javax.crypto.Cipher: int doFinal (java.nio.ByteBuffer, java.nio.ByteBuffer)

25. javax.crypto.Cipher: javax.crypto.ExemptionMechanism getExemption-Mechanism()

26. javax.crypto.Cipher: byte[] doFinal()

27. javax.crypto.Cipher: int doFinal (byte[], int, int, byte[], int)

28. javax.crypto.ExemptionMechanism: javax.crypto.ExemptionMechanism getInstance (String)

29. javax.crypto.Cipher: javax.crypto.Cipher getInstance (String, Provider)

30. javax.crypto.Cipher: int getBlockSize()

31. javax.crypto.Cipher: int getOutputSize(int)

32. javax.crypto.Cipher: void init (int, Key,
    AlgorithmParameters, SecureRandom)

33. Signature: Signature
    getInstance (String, Provider)

34. javax.crypto.Cipher: javax.crypto.Cipher
    getInstance (String, String)

35. javax.crypto.Cipher: void init (int, cert.Certificate)

36. Signature: Signature getInstance (String, String)

Bibliography

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, 2007.

[2] Amazon. Amazon Web Services. `http://aws.amazon.com/`.

[3] Apache. Harmony Java Virtual Machine. `http://harmony.apache.org/`.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, page 86, 1995.

[5] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002.

[6] Michael D. Bond, Varun Srivastava, Kathryn S. McKinley, and Vitaly Shmatikov. Efficient, context-sensitive detection of semantic attacks. In *UT Austin Computer Sciences Technical Report TR-09-14 2009*, 2009.

[7] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, 2002.

[8] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference and Exposition*, 2:1119, 2000.

[9] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 435–445, 2007.

[10] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 109, 1999.

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, 2001.

[12] FindBugs. FindBugs Bug Description. `http://findbugs.sourceforge.net/bugDescriptions.html`.

[13] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th international conference on Software Engineering*, pages 458–467, 2007.

[14] Gartner. Gartner Says Security Delivered as a Cloud-Based Service Will More Than Triple in Many Segments by 2013. `http://www.gartner.com/it/page.jsp?id=722307`.

[15] GNU. The GNU Compiler for the JavaTM Programming Language. `http://gcc.gnu.org/java/`.

[16] GNU Classpath. Classpath Java Virtual Machine. `http://www.gnu.org/software/classpath/home.html`.

[17] Google. Dalvik Virtual Machine. `http://www.dalvikvm.com/`.

[18] Google. Google Apps. `http://www.google.com/apps/`.

[19] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[20] IBM. Cloud Computing. `http://www.ibm.com/developerworks/cloud/`.

[21] IBM. IBM Point of View:Security and Cloud Computing. `ftp://public.dhe.ibm.com/common/ssi/sa/wh/n/tiw14045usen/TIW14045USEN_HR.PDF`.

[22] Jnode. Jnode Project. `http://www.jnode.org/`.

[23] Kaffe. Kaffe Project. http://www.kaffe.org/.

[24] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, 2006.

[25] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, 2001.

[26] Antonio Menezes Leito. Detection of redundant code using r2d2. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:183, 2003.

[27] Mono. Mono Project. http://www.mono-project.com/.

[28] Oracle. Sun Java Vitual Machine. http://java.sun.com/.

[29] Oracle. Java Compatibility, 2001. http://java.sun.com/developer/technicalArticles/JCPtools/.

[30] Oracle. Java Security Architecture, 2010. http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html.

[31] Salesforce. Salesforce Platform. http://www.salesforce.com/platform/.

[32] A. Prasad Sistla, V. N. Venkatakrishnan, Michelle Zhou, and Hilary Branske. Cmv: automatic verification of complete mediation for java

virtual machines. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 100–111, 2008.

[33] SOOT. Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[34] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: automatically inferring security specifications and detecting violations. In *Proceedings of the 17th conference on Security symposium*, pages 379–394, 2008.

[35] Tim Mather and Subra Kumaraswamy and Shahed Latif. *Cloud Security and Privacy*. O'Reilly Media, 2009.

[36] Varun Srivastava. The vulnerabilities submitted to Classpath. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42390.

[37] Varun Srivastava. The vulnerabilities submitted to Harmony. https://issues.apache.org/jira/browse/HARMONY-6367.

[38] Varun Srivastava. The vulnerabilities submitted to Sun. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6914460.

[39] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 218–228, July 2002.

[40] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[41] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, 2002.

Vita

Varun Srivastava was born in Allahabad, India on 27 December 1980, the son of Mr. Virendra Pratap Srivastava and Shashi Srivastava. He received the Bachelor of Technology degree in Information Technology from the Indian Institute of Information Technology, Allahabad in 2004. He worked in Research and Development departments of Hughes Software Systems, IXIA and Citrix between 2004-2008, as Software Development Engineer. In August, 2008, he entered The Department of Computer Sciences at The University of Texas at Austin.

Permanent address: 1905 Nueces Street, Suite 304
Austin, Texas 78705

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.