CMV: Automatic Verification of Complete Mediation for Java Virtual Machines[*]

A. Prasad Sistla V.N.Venkatakrishnan Michelle Zhou Hilary Branske
Department of Computer Science
University of Illinois at Chicago
sistla, venkat, yzhou, hbranske@cs.uic.edu

ABSTRACT

Runtime monitoring systems play an important role in system security, and verification efforts that ensure that these systems satisfy certain desirable security properties are growing in importance. One such security property is complete mediation, which requires that sensitive operations are performed by a piece of code only after the monitoring system authorizes these actions. In this paper, we describe a verification technique that is designed to check for the satisfaction of this property directly on code from Java standard libraries. We describe a tool CMV that implements this technique and automatically checks shrink-wrapped Java bytecode for the complete mediation property. Experimental results on running our tool over several thousands of lines of bytecode from the Java libraries suggest that our approach is scalable, and leads to a very significant reduction in human efforts required for system verification.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls Verification*

General Terms

Security, Verification

Keywords

Runtime Monitoring, Complete Mediation

1. INTRODUCTION

Systems that perform runtime monitoring for enforcing security properties play an important role in securing computing infrastructures. A wide variety of computing systems employ runtime monitoring mechanisms: operating systems, authorization systems, language interpreters are some key examples. Research efforts to verify the correctness of the implementation of these systems are of paramount importance, as these efforts ensure that these systems are trustworthy, and are therefore of significant research interest.

The Java programming platform is an instance of a system that employs runtime monitoring. Designed to run untrusted mobile programs, the Java environment has achieved widespread acceptance due to reasons of platform independence and security. Platform-independence was achieved by proposing a platform independent bytecode format executed by a platform-specific Java virtual machine (JVM). Security was achieved providing a customizable security architecture for executing these applications. The implementation of the security mechanism was achieved through runtime monitoring of accesses [16] to system resources such as the filesystem and networking.

Security in the JVM is designed to handle code that may come from different *code sources*, and a customizable policy assigns each of these code sources with a set of permissions. The mapping of code sources to different permission sets allows the JVM to mediate accesses based on this policy. The enforcement of this authorization policy cannot be left to the underlying operating system, which has no context to distinguish between operations done by the JVM from the operations done by the applications running above it.

The JVM thus is responsible for mediating access to operations to various underlying resources. The operations that implement this mediation are implemented in the Java system libraries. Although specifics vary for each virtual machine implementation, typical virtual machines implement the security sensitive operations that access resources as *native* methods (i.e., those implemented in platform dependent code) in the corresponding library class. Mediation to such resources is done through calls to the JVM security subsystem, by calling the `SecurityManager` class. [1]

A typical piece of code that calls the `SecurityManager` is:

```
public FileInputStream(FileDescriptor fdObj) {
   SecurityManager security =
                   System.getSecurityManager();
    if (security != null) {
        security.checkRead(fdObj); }
    ... /* sensitive native operation */
}
```

[*]This research is supported in part by NSF grants (CNS-0716584), (CNS-0551660) and (CCF-0742686).

[1]In recent versions of the JVM, the `SecurityManager` is a wrapper class for another class called the `AccessController`, which implements the stack inspection procedure. We do not distinguish between the `SecurityManager` and the AccessController in this paper.

This code sequence from the `FileInputStream` class looks for an instantiated `SecurityManager` object. If one exists, it calls the `SecurityManager`, which then checks whether the current calling sequence has privileges to perform the file read operation.

It is worth noting that a JVM that runs on a browser always has an instantiated `SecurityManager` (for applets), and therefore security checks are enabled by default. In contrast, for a stand-alone VM, only the security subsystem is invoked (typically through a command-line switch to the Java interpreter), the JVM mediates policy based access to sensitive resources.

Whenever a native operation is invoked, the code performs a call to the `SecurityManager` that employs a procedure called *stack inspection* which checks whether the current code context (i.e., the different code sources in the calling sequence) has the permission to perform the given operation.

The trustworthiness of the implementation of the stack inspection mechanism depends on the validity of two key properties of the implementation:

- *Complete mediation* The security manager is consulted in any request from the application code that leads to execution of a security sensitive operation.

- *Stack inspection implementation correctness* At runtime, the security manager correctly implements the stack inspection procedure by checking whether each of the code sources in the calling sequence has the requested permission.

Assurance about the latter property requires verifying the correctness properties of the implementation of the stack inspection. While this is theoretically feasible, fully automated verification of this piece of stock code is limited by challenges in current formal verification technology. A more practical approach is to verify models (obtained by abstraction of system behavior) of the stack inspection procedure which are more amenable to verification. One such effort was carried out by using a belief-logic based approach by Wallach and Felten [30]. Since the code that implements stack inspection is localized in a set of methods, one can employ human assisted methods for verification that involve some automated components such as theorem provers (such as [23]).

The former property, known as *complete mediation* [25], is equally important in the context of assuring trustworthiness of JVM security. More specifically, we can state the property as follows:

Property 1. (Complete Mediation) *When the* `SecurityManager` *is instantiated, a security sensitive operation is never performed without consulting the* `SecurityManager`*.*

The focus of this paper is on *automated verification* methods that provide assurance about the satisfaction of the complete mediation property by (an implementation of) a Java virtual machine.

Although the system call interface in an operating system presents a view that the JVM presents to Java application programs, verification of the complete mediation property for the system call interface is a relatively easier task. This is because the system call interface *localizes* the authorization checks on the system call at the system call boundary, and hardware-based support guarantees that the application can transfer to the kernel only through this boundary. Since the checks are localized (in the code at the boundary), manual verification efforts are a possibility. In contrast, in a JVM like implementation, each of the individual library classes performs sensitive native operations, and guards these native operations through security checks. In a JVM, several hundreds of methods perform these operations, and enforcement is done purely at the software layers. Any manual efforts that require verification of several thousands of lines of code (across hundreds of methods) spanned by these operations would be tedious if not impossible, and their correctness would be unreliable. Automatically guaranteeing that the authorization operations cannot be bypassed (due to the absence of checks) is therefore a critical problem.

There have been a number of efforts to verify similar security properties in the security community [4, 6, 27, 2]. This is, to the best of our knowledge, the first effort in providing large-scale verification assurance about complete mediation for Java library classes. In this paper, we discuss and evaluate JVM implementations for UNIX-like operating systems (such as Solaris, Linux, FreeBSD), where resource access operations are implemented in the JVM as native methods using POSIX-style system calls or library functions.

Verifying the complete mediation property for the JVM is actually a problem in *open system* verification. An open system is one which exports a set of methods that is called by the environment (i.e., client application code). The code for the environment itself is not available, unlike a *closed system* where the entire system code (i.e., libraries plus the application code) is available. Therefore techniques that address such a problem must assume an all powerful adversary capable of making any sequence of calls to security sensitive operations in the library. Specifically in case of the JVM, the problem is about designing an adversary that is capable of making accesses to security sensitive operations without being mediated by a `SecurityManager`.

In this paper, we describe the following contributions:

- We present a scalable model checking technique for this open system verification problem. This technique automatically checks whether there are paths in a VM implementation that will allow an application class to access a sensitive resource bypassing the security checks. The heart of our approach is based on a compositional checking procedure that computes method summaries.

- We implemented our algorithm in a tool that we call CMV (Complete Mediation Verifier) to directly analyze bytecode implementation of classes.

- We present an experimental evaluation of our tool with two highly used JVMs, the HotSpot VM (from SUN Microsystems) and the Harmony VM (from Apache Foundation).

This paper is organized as follows: We survey related work in Section 2. Section 3 presents our main technical contributions including illustrations of the key aspects of the problem and the formalism behind approach. Section 4 discusses the details of the implementation of this approach. Experimental results are discussed in Section 5. We conclude in Section 6.

2. RELATED WORK

Java was built as a platform for secure execution of mobile code, and it employs runtime monitoring [16] for authorizing access to resources. It is to be noted that there is extensive work on frameworks for ensuring mobile code security such as proof-carrying code [22], model-carrying code [26] and proof-linking [12].

Model Checking The problem of verifying complete mediation in a runtime monitoring system such as Java can be expressed as a problem in model checking [7]. By expressing the complete mediation problem in temporal logic, one can indeed use general purpose model checkers [4, 17, 18, 8, 5, 6] for checking this property. The advantage of general purpose model checking tools is that they handle verification of arbitrary user-specified properties that are expressible in temporal logic like formalisms. In this paper, the techniques we have developed have been specialized to the complete mediation property. Property specific customization enables the technique to be scalable to a large code base such as the Java system libraries.

The research effort closest to our work is by Jensen et al. [19]. Their work also gives model checking based methods for checking security properties in Java code. In this approach, the property to be verified is specified in a temporal logic interpreted over the calling stack sequence. In their system, we can express the authorization property by asserting that every caller on the called stack has the permission to execute the current sensitive operation, thus enabling static verification of permission checking for the given set of classes. However, using their approach one cannot assert that every sensitive operation is preceded by a security check, because the security check operation returns (and is no longer in the call stack) when the sensitive operation is performed. Since their work requires the complete calling sequence for verification of the property, they can only check *closed* systems (i.e., including all application code). In contrast, our approach is intended for verifying *open* systems, which further implies that every closed system that employs this verified library will satisfy the complete mediation property.

Static Analysis Several static analysis techniques have been used in the past in large scale bug finding [9, 27, 21] and for access rights analysis [20]. Closely related to our work are works by Zhang et al. [31] and by Fraser et al. [13]. [31, 13] both use type qualifiers to check the authorization properties in C programs. This involves introducing type qualifiers for arguments to sensitive operations, and the system checks these properties using type qualifier inference. They demonstrated their work over similar large code bases such as the SE-Linux and MINIX kernels. However, their approach assumes that there are variables that are common to sensitive operations and security checks, and in several instances in Java code base this assumption does not hold. Although one can modify the programs by introducing additional variables (as was done in [31] to avoid limitations with flow insensitive type qualifiers), this may introduce large changes to programs. For instance, if the security checks and sensitive operations are present in different methods, this may require changes to the type signatures of all the methods in the called sequence and /or duplication of code of methods.

Retrofitting code for authorization There have been efforts for retrofitting code for enforcing authorization properties, for Java [11, 10, 29] as well as for servers written in C that perform authorization [14]. Naccio [11] is a system for Java code that takes an abstract description of resources and permissions and generates code that enforces this property. Retrofitting code is a complimentary effort, that enables the end user to enforce authorization properties on source code. In contrast, ours is a verification effort that checks shrink-wrapped software code distributed by software vendors. In fact, retrofitting needs to be employed only in the situations when verification efforts like ours identify possible unsafe methods.

3. OUR APPROACH

3.1 An illustrative example

In this section, we present a running example that we will use to illustrate the main ideas. This example shown in Figure 1 involves three illustrative example methods $Meth\_X$, $Meth\_Y$ and $Meth\_Z$ (or simply X, Y and Z) whose code was modeled based on methods in the Java standard library. These methods perform the (native) call openFileOrDir, which is a sensitive operation, and another two sensitive operations fsStatFile, fsStatDirectory. All the three sensitive operations require FilePermission to read the file associated with the pathname parameter.

The idiom for consulting the security manager is given in the code for Y and Z. After getting the current `SecurityManager` object, the code checks if it is initialized (non-null). The checkPermission method proceeds to check whether all the callers in the calling sequence have `FilePermission` to read the file associated with the pathname, required for executing any of the sensitive operations: `openFileOrDir`, `fsStatFile`, `fsStatDirectory`.

Notice that X is the only public method in the library. Since X calls Y or Z, and subsequently performs `openFileOrDir`, it is important to ensure that the security manager is consulted in every path that leads to this operation. This property holds in the path that contains the call to Y. However, the `SecurityManager` is not consulted in one of the paths after Z is called, as the `else` branch in Z does not consult the security manager.

In this paper, we focus on the design and implementation of an automated analysis technique to identify such paths by statically analyzing Java code. We want to design a procedure that will, when given a method M, analyze the method (and all the methods called by M in its transitive closure) and ask if such paths exist. Since we propose using a sound analysis for this problem, when the analysis procedure gives a "no" answer then this definitely implies that there are no such bad behaviors. If the analysis procedure gives a "yes" answer then there may or may not be such a computation. In this case, the analysis can output a sequence of statements, a possible *witness* (sometimes called a counter example), whose execution may create such a computation. A more detailed, possibly manual or semi-automatic, examination of the witness can be done to check if the path is feasible in practice (in this case a false alarm), or ascertain if this is a security vulnerability.

We point out that use of automatic program analysis to identify potentially unsafe methods, and using human assisted methods with the help of the witness generated can result in considerable savings of human efforts. In addition,

```
public void Meth_X(
      String pathname, int x) {
0:  if ( x == 0){
1:      Meth_Y(pathname);
    }
    else {
2:      Meth_Z(pathname);
    }
    //sensitive-op
3:  openFileOrDir(pathname);
}
```

```
private void Meth_Y(String pathname) {
0:   SecurityManager sm =
              System.getSecurityManager();
1:   if (sm != null) {
2:       sm.checkPermission(FilePermission
              (pathname, "read"));
     }
3:   fsStatFile(pathname); //sensitive-op
}
```

```
private void Meth_Z(String pathname) {
0:     if (pathname.endsWith("/")){
1:         SecurityManager sm =
                System.getSecurityManager();
2:         if (sm != null) {
3:         sm.checkPermission(FilePermission
                (pathname, "read"));
           }
           // sensitive-op
4:         fsStatDirectory(pathname);
       }
       else {
5:         // other operations
       }
}
```

Figure 1: An example that involves sensitive operations

the absence of any witnesses can be used to *certify* that the system possesses the complete mediation property. In our experiments, we have been able to analyze methods in the order of several thousands of lines of code, while keeping the false positives in the order of tens of methods. Specialized techniques can be developed for addressing these false positives, and careful manual effort can be employed to rule out bugs.

3.2 Challenges for Verification

Verifying the software implementation of the Java access control is a important problem. The main challenge is to design a procedure that scales to several thousands of methods present in the JVM libraries. Scalability demands the following two characteristics in any solution.

- *Compositionality* In the above example, in the context of analyzing method X, we have to analyze methods Y and Z. In some other context, we may need to analyze methods Y and Z. In this case, we should be able to reuse the results of the first analysis. This property of being able to (re)use the results of the analysis of individual methods, is the compositionality property, and is a key concern in the design of scalable methods for security verification.
- *Low Complexity* A naive approach that looks for control paths in all possible method sequences in the library will fail due to the enormous size of the Java library, which has several thousands of methods. The space complexity of such a naive control flow analysis can be exponential in the sum of the sizes of the methods in the Java library, as explained later. For the size of the Java library, even polynomial methods are not good enough, as cubic or even quadratic algorithms can be expensive.

The procedure we describe in the next section satisfies these two important properties.

3.3 Technical Approach

Problem Scope We assume that we are given the set of security sensitive operations and the Java permissions required for performing these operations. We also assume that we are given a set of method definitions that is self contained, i.e., the definition (body code) of all methods invoked by any method is also present in the set. The only exception is the source for sensitive native methods, which we do not analyze.

For each *public* method M among them, we want to check the following property: *does there exist a call sequence of methods starting with M, that invokes a sensitive operation without consulting the* `SecurityManager` *before the operation?* Since the method is `public`, it can be called from any application class, and therefore we call such a method M a *risky* method. Our task is to identify such risky methods in the Java standard library. In the rest of this section, we introduce our approach of computing these summaries through gradual refinement of ideas.

A first approach To identify such paths we can statically analyze the code of library classes. A first approach to verifying whether a given method M that operates on a sensitive resource respects complete mediation is to start with the control flow graph (CFG) of the method and look for paths in it that lead to the sensitive operation without calling the `SecurityManager`. Figure 2 gives the CFGs for methods X, Y, and Z presented in Figure 1.

To handle procedure calls made by method M, we can extend the above method by extending the CFG of M with inline expansion of the CFGs of all the methods called by M. We can then analyze every path in the resulting graph (called the expanded CFG or ECFG(M)) that leads to a sensitive operation.

The above naive method may actually work for small code bases. However, it has two main drawbacks. The first drawback relates to the case where recursion is present among methods in the library, where it is well known that inline expansion will not work. For instance, the presence of two or more mutually recursive methods will result in an infinite expanded graph.

Secondly, due to the inline expansion, in general, even if there is no recursion, the size of $ECFG(M)$ can be exponential in the sum of sizes of the methods. Thus a method that constructs the $ECFG(M)$ is not scalable for large code bases. Our aim, therefore is to analyze the code base *without* constructing ECFGs.

Method Summaries To analyze the code base without constructing ECFGs, our approach needs to reuse the results of analysis of a method. In our approach, the results of analyzing each method is stored in its *method summary* (One of the earliest works that employed procedure summaries is [24]). Intuitively, the summary of a method M denotes the effect of a call to M by any other method for checking the presence of security checks. The heart of our approach involves a novel procedure for computing (and reusing) these summaries for verification.

We classify the nodes of the control flow graph of a method M into the following types: *entry* node, *return* node, *security check*, *sensitive operation*, *method invocation* and *neutral* node. The first five are self-explanatory, neutral nodes are those that do not fall into any of the first five types.

3.3.1 Computing summaries

The summary of a method M is a 2-tuple that contains two following components:

- *Presence of paths that do not have security checks* This component of the summary reflects the existence of paths from the entry node of M to a return node that do not contain any security check. If such a path exists in a method, then we include the *insecure_path* in the first component of $summary(M)$. We say that a method is *all_path_secure* if *insecure_path* is not in the first component of $summary(M)$.
- *Presence of unguarded sensitive operations* If there is a path in the expanded control flow graph from the entry node to a sensitive operation without the presence of a security check before that operation, then our approach will include the label *bad* in the second component of $summary(M)$. We call a method *good* if the label *bad* is not in the second component of $summary(M)$.

Thus we can have the following four types of summaries: $\langle insecure\_path, bad\rangle$, $\langle insecure\_path, \perp\rangle$, $\langle \perp, bad\rangle$, $\langle \perp, \perp\rangle$.

The method summaries of X, Y and Z in Figure 1 are given at the bottom of Figure 2. For instance, method Y is *all_path_secure* and *good*, and Z is *good* but not *all_path_secure*. Consequently, X, which calls Y and Z along different paths, is *bad*. We also wish to note that a method may be *bad*, but still can be *all_path_secure*; this is because it may contain a path having a sensitive operation that is followed by a security check.

For a method M, we define the boolean function $insecure\_path(M)$ to be true if it contains the label *insecure_path* in its first component. In a similar fashion, we can define the boolean function $bad(M)$. We can also define $good(M)$ (which is simply *not* $bad(M)$ and $all\_path\_secure(M)$ (which is same as *not* $insecure\_path(M)$).

Our objective is then to compute for each method the two components (corresponding to *insecure_path* and *bad*) of the summary. Our approach computes these summaries by analyzing the individual CFGs of these methods.

Observation 1 Let M_1 be a method for which $bad(M)$ is true. Let M_2 be a method that calls M_1. Then note that, we can effectively replace this method invocation node in $CFG(M_2)$ with a sensitive operation node while computing the *bad* summary of M_2 (i.e., the second component).

The above observation says that calling the method M_1 for which $bad(M_1)$ is true is effectively (i.e., for the purposes of computing bad summaries) equivalent to performing a sensitive operation. Note that calling M_1 does not however add *bad* to the summary of M_2 because there may have been a preceding security check in the body of M_2 before the call to M_1.

Observation 2 Let M_1 be a method for which $insecure\_path(M_1)$ is true. Let M_2 be a method that calls M_1. Then note that, we can effectively replace this method invocation node in $CFG(M_2)$ with a neutral node while computing the *insecure_path* summary of M_2 (i.e., the first component).

Observation 3 Let M_1 be a method for which $all\_path\_secure(M_1)$ is true and $bad(M_1)$ is false. Let M_2 be a method that calls M_1. Then note that, we can replace this method invocation node in $CFG(M_2)$ with a security check node in computing the *bad* summary of M_2 (i.e., the second component).

The above three observations allow us to compute the *bad* and *insecure_path* summaries of a set of methods easily. In the absence of recursion, we can sort the methods based on their reverse invocation sequence; (i.e., if M_2 calls M_1 then M_1 appears in earlier in the sequence). Such a sequence can be constructed using the reverse topological sort of the call graph of all methods.

The first method M_1 in this sequence calls no other methods. Therefore, we can compute its summary by analyzing all its paths. We move on to M_2, which is next in the reverse invocation sequence. While computing *insecure_path* summary of M_2, if $insecure\_path(M_1)$ is true, then we can effectively consider M_1 to be a neutral node (Observation 2). Similarly, while computing bad summaries of M_2, if $bad(M_1)$ is true, we can replace the call node to M_1 with a sensitive operation node (Observation 1). If $all\_path\_secure(M_1)$ is true, and $bad(M_1)$ is not true, we can replace the call node to M_1 with a security check (Observation 3). In a similar manner, we can successively compute the *insecure_path* and *bad* summaries for all the methods in the sequence, and then identify *bad* methods that are `public`. These methods are risky, as they provide a direct path from application code to a sensitive operation.

The above algorithm is also very efficient, and one can show that the total running time of this algorithm is linear in the sum of the size of the methods analyzed.

3.3.2 Handling recursion

Unfortunately, the approach that was described above will not work in the case of recursive calls. In the case with recursion, we will not be able to uniquely sort the methods in their reverse invocation order, as the call graph will contain cycles due to recursion.

The standard approach to deal with recursion involves iteratively computing summaries for the procedure until the computation converges to a fixed point. Such an approach will compute the summaries, but the worst case running time of this procedure will be of order $O(N * M)$ where N is the number of methods and M is the sum of the sizes of all the methods analyzed, thus making it highly inefficient.

3.3.3 A new efficient solution

We have devised a new solution that computes the summaries for the recursive case that is highly efficient. The running time of this new procedure is still linear in the sum

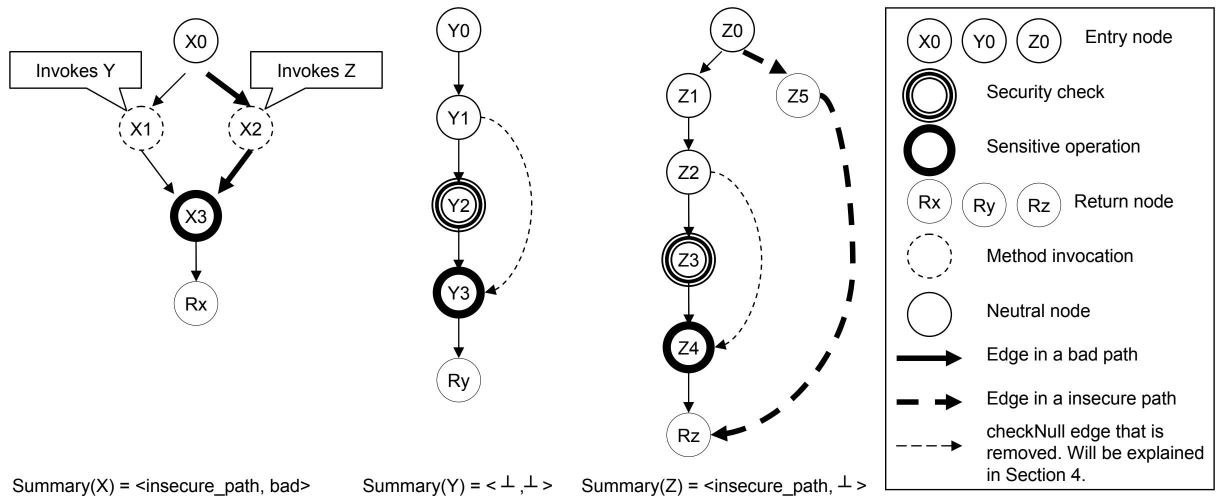

Figure 2: Control flow graphs for the example in Figure 1

of the sizes of methods analyzed, effectively equaling the running time of the algorithm for non-recursive case that was described earlier. We describe the procedure for computing the summary for a procedure below.

Checking for insecure paths To determine if method M has *insecure_path*, we perform a search of $CFG(M)$ checking for the existence of a path, that does not contain a security check from its entry node to its return node. This search starts from the entry node and operates as follows. Whenever we encounter a node that is a neutral node or is a sensitive operation, we continue the search from its successors. Whenever we encounter a security check node, we do not proceed beyond that node but explore other nodes. This is because the path we are looking for is an insecure path and therefore cannot contain a security check.

Whenever we encounter a method invocation node u that invokes method M', and *insecure_path* is in $summary(M')$, then we continue the search from the successor nodes of u. This is because there is a path without a security check, in the expanded control flow graph of M passing through M', from the entry node of M to u and its successors. On the other hand, if *insecure_path* is not in $summary(M')$, we put node u on a waiting queue, $WQ(M')$, associated with M'. Whenever M' is determined to have an insecure path, at that point, the search of $CFG(M)$ is resumed from node u. Whenever a return node is encountered in the search of $CFG(M)$, we include *insecure_path* in its summary and at this time we examine all entries in $WQ(M)$. For each node v on this queue, we do as follows. If *insecure_path* is already in the summary of the method containing v then we ignore v; otherwise, we continue the search from node v.

The above intuitive procedure is implemented as shown in Figure 3 and can be explained further as follows. For each node u of a control flow graph, we associate a binary flag *u.visited* which by default set to false and set to true when first encountered. We also assume that *u.successors* gives a list of successors of u, also *u.method* the name of the method to which u belongs. With each method M, we maintain a set $WQ(M)$ as indicated earlier. The search algorithm maintains a set data structure Q. Essentially, Q is the set of nodes that need to be explored. It is initialized to be the set of entry nodes of all the methods and each such node is marked as visited. The algorithm is self explanatory. The correctness of the algorithm of Figure 3 is stated by the following lemma.

Lemma 1:

1. For every node u placed on Q, there exists a path in $ECFG(u.method)$ from the entry node of *u.method* to u such that none of the nodes on the path (excepting u) is a security check.
2. For every method M, *insecure_path* is added to $summary(M)$ iff there is a path not containing a security check from the entry node of M to a return node of M in $ECFG(M)$.

Complexity Analysis In the above algorithm, it is easy to see that each node of a control flow graph is placed on Q at most once if it is a node other than a method invocation; a node which is a method invocation is placed on Q at most twice (the second time is when it is removed from $WQ(M')$ where M' is the method invoked by the node). Hence the complexity of the above algorithm is linear in the sum of the sizes of the control flow graphs of all the methods.

Algorithm for determining bad methods Now, we present the algorithm for determining bad methods, which is executed after the algorithm for insecure paths terminates. We perform a search of the $CFG(M)$, for each method M, looking for a path from the entry node to a sensitive operation without a security check before it. We use the same data structures as given above which are initialzed similarly. The search of the control flow graphs of all the methods is done at the same time using the set data structure Q. The only difference is in processing of a node u when it is removed from Q. Let us say that *u.method* is M. If u is a neutral node then search is continued from its successors. If u is a security check or a return node then it is ignored.

If u is a method invocation that invokes method M' then the following actions are taken. First observe that there is a path, not containing any security check, from the entry

```
Initialze();
For each method $M$
    Set $summary(M)$ to be $\langle\perp,\perp\rangle$;
While $Q \neq \emptyset$ {
    Remove a node $u$ from $Q$;
    If $insecure\_path(u.method)$ : Continue;
    Perform one of the following steps based on type of $u$
    $u$ is a sensitive operation or a neutral node :
        For each $v$ in $u.successors$
            If $v.visited = false$
                $v.visited := true$, add $v$ to $Q$;
    $u$ is a return node :
        add $insecure\_path$ to $summary(u.method)$,
        For each $v$ in $WQ(u.method)$
            if ! $insecure\_path(v.method)$ add $v$ to $Q$;
        set $WQ(u.method)$ to the empty set;
    $u$ is a method invocation to method $M'$ :
        If $insecure\_path(M')$
            For each $v$ in $u.successors$
                If $v.visited = false$
                    $v.visited := true$, add $v$ to $Q$;
        Else add $u$ to $WQ(M')$;
    $u$ is a security check : Continue;}

Procedure Initialize() {
    $Q := \emptyset$;
    For each method $M$
        $WQ(M) := \emptyset$;
        For each node $u$ in $CFG(M)$
            $u.visited := false$;
        Add the entry node $u$ of $M$ to $Q$,
        $u.visited := true$ ; }
```

Figure 3: Algorithm for computing *insecure_path* summaries

node of M to u. (If there were a check, then we would have skipped the check and therefore wouldn't have reached u.) So if M' is bad then M is also bad and is marked so. However, if M' is not currently marked as a bad, then we need to mark M as bad if and whenever M' is determined to be bad. To do this, we place node M on the waiting queue of M', which is $WQ(M')$ (note that $WQ(M')$ contains method names not nodes as is the case in case of the algorithm for determining *insecure_path*).

Further more, if M' han an insecure path then there is a path without a security check, in the expanded control graph of M, from the entry node of M, passing through nodes of $CFG(M')$, to the successors of u. Thus we need to continue the search beyond u for a bad path. Lastly, if u is a sensitive operation then this implies that there is a bad path from the entry node of M to u. In this case, we add the label *bad* to $summary(M)$. In this case, we need to mark all the methods waiting on M directly or transitively as *bad*. This is accomplished by the procedure $MarkAsBad()$ which invokes procedure *Compute_Waiting*. $Compute\_Waiting(M)$ (whose code is not given) computes all the methods that are directly or transitively waiting on M by taking a union of all methods in $WQ(M)$ and all the methods that are waiting on the methods in $WQ(M)$ and so on. The soundness of the algorithm is stated by the following lemma.

```
Initialze();             // same as in figure 3
while $Q \neq \emptyset$ {
    Remove a node $u$ from $Q$;
    If $bad(u.method)$ : Continue;
    Perform one of the following steps based on type of $u$
    $u$ is a neutral node :
        For each $v$ in $u.successors$
            If $v.visited = false$
                $v.visited := true$, add $v$ to $Q$;
    $u$ is a sensitive operation :
        $MarkAsBad(u)$;
    $u$ is a method invocation to method $M'$ :
        If $bad(M')$
            $MarkAsBad(u)$,
            Continue;
        Else add $u.method$ to $WQ(M')$;
        If $insecure\_path(M')$
            For each $v$ in $u.successors$
                If $v.visited = false$
                    $v.visited := true$, add $v$ to $Q$;
    $u$ is a security check or a return node : Continue;}

Procedure $MarkAsBad(u)${
    $Z = Compute\_Waiting(u.method) \cup \{u.method\}$;
    For each $M$ in $Z$
        add $bad$ to $summary(M)$,
        set $WQ(M)$ to the empty set;}
```

Figure 4: Algorithm for computing *bad* summaries

Lemma 2:

1. For every node u placed on Q, there exists a path in $ECFG(u.method)$ from the entry node of $u.method$ to the u such that none of the nodes on the path (excepting u) is either a security check or a sensitive operation.
2. For any method M, label *bad* is added to $summary(M)$ iff there is a path without any security checks from the entry node of M to a sensitive operation in $ECFG(M)$.

Complexity The complexity of the algorithm is linear in the sum of the sizes of the control flow graphs of all the methods. To see this, observe that each node in the control flow graph of a method is added to Q at most once. Also the sum of the sizes of $WQ(M)$ for all M is bounded by the number of method invocation nodes in all the methods. Also, each entry in $WQ(M)$ is processed at most once, that is when the label *bad* is added to $summary(M)$. Thus, we see that the complexity of the algorithm is linear in the sum of the sizes of all the control flow graphs.

Modification to compute summaries on-the-fly In the above presentation of the approach, we first execute the algorithm that computes *insecure_path* summaries for every method and then execute the algorithm for identifying bad public methods. However we can modify the approach so that we first invoke the algorithm for identifying bad public methods, and during its execution, invoke the algorithm for computing *insecure_path* summaries for each method on a demand driven basis. This modified approach avoids unnecessary checks for insecure paths.

A more detailed comparison to SLAM The Bebop model checker [3] of the SLAM toolkit [4] is a general purpose model checker for verifying temporal properties of binary programs with procedures and has complexity linear in the sum of sizes of the control flow graphs of the methods. Bebop tracks the data values in addition to analyzing the control flow graphs, while we only deal with control flow graphs. Bebop also computes summaries during the model checking process in the form of summary edges. The summary edges are associated with method invocation nodes, i.e., once for each call. On the other hand, our summaries are associated with methods. Since the same method may be called from multiple points, associating summaries with methods is more efficient. Further more, call nodes which are preceded by security checks are never processed in our method.

Generating a witness If a method M is bad, then a witness for this can be computed by modifying the above algorithm as follows. In this case, a path leading to a node performing a sensitive operation is output as a *compressed witness* when such a node is reached in the search of $CFG(M)$. This is achieved by maintaining a *parent* pointer with each node u that points to the node from which u is visited first. The path leading to the node u from the entry node of M is obtained by travesring along the *parent* pointers. (Note that the length of such a path is bounded by the size of the method M.) A similar witness is computed if $insecure\_path(M)$ is true. In this case the witness is a path starting with the entry node of M and ending with a return node of M. We call these paths as compressed witnesses since they may contain nodes which are method invocations. If a method M' is invoked at a node on a compressed witness, then it has to be the case that $insecure\_path(M')$ is true. For each such method M', we would have generated a similar compressed witness for M' when its summary was computed. Since only one compressed witness is necessary for each method that is *bad* and one such witness for each method that is has *insecure_path*, the sum of the sizes of all such compressed witnesses is only linear in the sum of the sizes of all the methods. We can show that the complexity of the resulting algorithm is still linear in the sum of the sizes of all the methods. Using such compressed witnesses, an actual witness path in the extended control flow graph can be generated; such a path can be of exponential length and needs not be generated explicitly for debugging purposes.

Handling multiple permissions So far we have only dealt with the case of a single sensitive resource. However, there are multiple sensitive resources in a JVM such as files and network operations, and Java security checks (such as `checkRead` (for `FilePermission`) and `checkConnect` (for `SocketPermission`)) corresponding to these resources. Therefore we modify the summaries to store the security check that was used to compute these summaries. In addition, if a method summary includes a security check that we are not currently looking for, we simply ignore this security check node and treat it as a neutral node. For instance, if we are checking the `File` resources, and encountering a method M whose summary represents its paths that contain the `SocketPermission` security check, then we treat any call to this method as a neutral node.

4. IMPLEMENTATION

Our implementation is based on bytecode analysis and transformation using the Soot framework [28]. Bytecode verification has the advantage that it retains the high level structure of the source code, while allowing us to directly verify stock library bytecode as distributed by the software vendors.

Identifying Sensitive operations. Our work in this paper describes a technique that checks for the satisfaction of the complete mediation property, when given a set of sensitive operations and their corresponding security checks. Identifying these sensitive operations in code is an orthogonal problem, that has been studied using techniques such as specification mining [1], and more recently, using concept analysis [15].

Our experimental evaluation considered sensitive operations that were implemented as `native` methods. For instance, in the HotSpot VM, such sensitive operations are implemented differently for each OS platform, and we have analyzed the implementation for the Linux and Solaris operating systems. We first identified the set of native operations directly from the type signatures of these methods. We further manually examined the native code to these methods, by looking at the C source code implementation, to ascertain whether these perform sensitive operations such as system calls. The above step completely identifies all sensitive operations that access resources that are managed by the underlying operating systems.

However, there are other forms of sensitive resources that are purely manged by the Java virtual machine. For instance, any property such as `java.vendor` is a resource managed internally in the JVM, and access to this resource requires the corresponding `java.util.PropertyPermission` from the calling context. For the purpose of analyzing code that accesses these additional resources, our technique can be augmented with specification mining techniques such as those described in [1, 15].

Filtering non-sensitive classes. For the sake of efficiency, we can exclude certain classes that do not handle sensitive resources. For this purpose, we wrote a scanner that inspects a class for the presence of the `SecurityManager` (or `AccessController`) or sensitive operations in any of its methods. If the above condition is not met, then the scanner concludes that the class does not manage sensitive resources, and therefore filters this class from the analysis. The scanner works on the premise that most of the Java library code is correct, and this code can be used to identify sensitive resources managed by it. Note that the filtered class may still call public methods of other classes that handle sensitive resources; in this case, this class is similar to an client application class from a verification point of view.

One exception to the above scanning condition occurs in the case of inner member classes. For instance, if class B is a inner member class of class A, then we ignore class B only if class A is being ignored.

Pruning the CFG for analysis For each method, we first construct its CFG, where each line of bytecode in the method is a node in the CFG. Several pre-processing pruning steps are done to the CFG before it is analyzed. The first case concerns the paths that do not need to be analyzed. In every call to the `SecurityManager`, there is a check for the security manager being non-null (as shown in the code examples in Figure 1). Since the complete mediation property (Property 1), requires that security manager be non-null, we delete the checkNull edge (and therefore the path) that checks for the `SecurityManager` being null before starting our analysis.

Privileged Operations Java allows a form of privilege escalation through the use of the `doPrivileged {S}` construct where `S` is a code segment that performs privileged operations. In this case, any security check performed inside `S` only considers the code context since the doPrivileged command is executed. Since we are only checking the (trusted) libraries, this means that the code context since the call to the doPrivileged operation has all the permissions to perform sensitive operations inside `S`. Also any security checks performed inside `S` cannot guarantee that sensitive operations, performed after `S` outside the scope of the doPrivileged method, have the required permissions. Hence we can ignore any security checks performed inside `S`. Thus, we handle the above doPrivileged command by replacing it with an invocation of a method M, whose body is `S`, and which is *good* and is not *all path secure*.

Significance of Context information Consider the invocation of a public method M on an object O by the application program. The object O must have been constructed before this invocation of M by executing a constructor for its class. It is possible that this constructor method performed a security check that involves the same permission required for the currently requested sensitive operation. One might be tempted to think that this `SecurityManager` check is enough for performing sensitive operations inside M. However, in general, this is not correct. This is because the code context inside M may be different from the code context when the security check was invoked inside the object constructor; for example, after constructing object O, the application program may have invoked code from other code sources that are active at the time when M was invoked and hence at the time the sensitive operation was performed. Thus any sensitive operation performed in the execution of a public method M should be preceded by a security check before the operation, but after the invocation of the method M. Our algorithm and the implementation have been designed keeping in mind the significance of this context information.

4.1 Other implementation issues

In this section, we discuss the limitations of our current prototype implementation and we discuss concrete ideas for improving the prototype to handle exceptions and method overriding.

Exceptions Exceptions can trigger special paths, in the program, resulting in additional paths from each method call instruction to the nearest enclosing catch block or to the caller. For handling exceptions, the standard control flow graph that we have needs to be modified in the following ways. For each method call, additional edges need to be created. An edge needs to be introduced from the method call node to the catch block enclosing the call node, if such a block exists. Furthermore, an edge is introduced from the method invocation node to a new return node that returns to the calling method of the current procedure being analyzed. This is to model an exception not handled by the enclosing catch block. Similarly, for any `throw` statement node enclosed in a `try` block, we add a new edge to the enclosing `catch` block. Also, if there are any `rethrow` nodes in the enclosing `catch` block, we will replace that node by a `return` node.

Once these additional edges are introduced, then our original method for computing summaries to identify risky methods will still be applicable.

Method Overriding Recall that a method of a base class can be overridden by the corresponding method of a subclass. The choice of this method is based on the object's type, which is only available at runtime. We propose a simple transformation to the method of the base class. Specifically, let us say M is the method of the base class being overloaded, and M_1, M_2, $\ldots$, M_n are overloading methods in subclasses. Then we can transform the code of M by adding the n-way non-deterministic branch statement at the start of the body of M, that makes a call to each one of these overriding methods. Once this branch statement is introduced, our original method for computing summaries will still be applicable in the presence of method overriding.

Making the above transformations to the methods and CFGs are simple and straightforward, which will be incorporated in a subsequent version of our prototype. In the following section, we present the experimental results obtained with the current prototype.

5. EXPERIMENTAL EVALUATION

In this section, we present our the results of the experiments that were performed with CMV. Most of CMV is written in Java code using the API provided by Soot, while some of the support scripts used in it are written in Perl.

Experimental Setup Two commercial off-the-shelf Java VMs were experimented in our verification effort using CMV. They are the HotSpot JVM, distributed by SUN Microsystems, and the Harmony VM, an open-source JVM produced by Apache Foundation. While the former is the most popular JVM implementation, the latter is being used in several industrial strength projects by the Apache Foundation. Verification efforts on these VMs therefore benefit several million end-users of these VMs. We verified the input output and network (java.io and java.net) subsystems, and the `Class` class in java.lang subsystem on both these VMs. These subsystems have several classes, and several hundreds of methods.

The methodology employed in our experiment was as follows: We run CMV on a class, which in turn calls Soot to analyze the class in whole program mode, which enables analysis of all methods called by methods of this class. The results of this analysis are summarized in a table, which is consulted when analyzing other classes and methods subsequently. The results were taken on a machine running Ubuntu distribution of GNU / Linux, on a machine running the AMD Athlon processor 2Ghz processor with 2GB physical memory.

Results Figure 5 gives the set of results for the HotSpot VM, and Figure 6 for Harmony VM. The second column identifies the package that the class reported in the first column belongs to. The third column gives the number of methods defined in the class. The fourth column reports the number of concrete methods (i.e. non-abstract and non-native methods). This is the number of methods in this class

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
JVM	Java	# Meth. in		Total					New	Real
class	package	JVM class	Concrete	Meth.	LOBC	Good	APS	Risky	Risky	Risky
File	java.io	54	54	708	12058	650	53	10	10	0
FileInputStream	java.io	17	10	40	459	40	3	0	0	0
FileOutputStream	java.io	18	12	42	486	42	5	0	0	0
ObjectInputStream	java.io	68	65	219	3748	197	3	5	3	0
ObjectOutputStream	java.io	59	57	146	2302	146	4	0	0	0
RandomAccessFile	java.io	47	36	66	775	66	2	0	0	0
Authenticator	java.net	15	15	21	204	21	4	0	0	0
CookieHandler	java.net	5	3	11	160	11	1	0	0	0
DatagramSocket	java.net	40	40	151	2047	123	10	23	22	0
HttpURLConnection	java.net	19	17	23	231	23	1	0	0	0
InetAddress	java.net	42	41	152	1875	148	5	1	0	0
MulticastSocket	java.net	18	18	583	9018	520	41	26	14	0
NetworkInterface	java.net	19	14	542	8194	498	34	9	0	0
ProxySelector	java.net	6	4	45	540	44	5	1	0	0
ResponseCache	java.net	5	3	11	160	11	1	0	0	0
ServerSocket	java.net	32	32	158	1932	154	15	1	0	0
Socket	java.net	61	61	710	11033	664	40	9	0	0
SocksSocketImpl	java.net	25	25	679	11085	633	41	9	0	0
URLClassLoader	java.net	17	17	539	8021	496	42	10	1	0
URLConnection	java.net	60	59	612	9128	569	41	9	0	0
URL	java.net	36	36	682	11663	624	38	10	0	0
java.lang.Class	java.lang	112	84	758	11049	699	49	21	11	0

Figure 5: Experimental Results from HotSpot VM

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
	Java	# Meth. in		Total					New	Real
JVM class	package	JVM class	Concrete	Meth.	LOBC	Good	APS	Risky	Risky	Risky
File	java.io	76	53	3076	53506	3075	121	0	0	0
FileInputStream	java.io	13	13	2929	50340	2928	105	0	0	0
FileOutputStream	java.io	13	13	2937	50396	2936	110	0	0	0
ObjectInputStream	java.io	84	74	3001	51885	3000	106	0	0	0
ObjectOutputStream	java.io	71	62	2989	51499	2988	106	0	0	0
RandomAccessFile	java.io	42	42	2974	51173	2973	106	0	0	0
Authenticator	java.net	15	15	2942	50428	2941	109	0	0	0
CookieHandler	java.net	6	4	2929	50297	2928	106	0	0	0
DatagramSocket	java.net	39	39	3029	51627	3027	114	0	0	0
HttpURLConnection	java.net	17	15	2948	50637	2947	108	0	0	0
InetAddress	java.net	60	53	3090	53838	3087	105	0	0	0
MulticastSocket	java.net	19	19	3026	51636	3024	118	0	0	0
NetworkInterface	java.net	13	12	2934	50405	2933	105	0	0	0
ProxySelector	java.net	6	4	2929	50300	2928	106	0	0	0
ResponseCache	java.net	8	6	2931	50302	2930	105	0	0	0
ServerSocket	java.net	29	29	3039	51696	3037	117	0	0	0
Socket	java.net	58	58	3091	52442	3089	118	0	0	0
URLClassLoader	java.net	28	28	3169	54427	3168	107	0	0	0
URLConnection	java.net	53	52	2987	50955	2986	111	0	0	0
URL	java.net	36	36	3024	52574	3023	107	0	0	0
java.lang.Class	java.net	62	62	65	163	65	0	0	0	0

Figure 6: Experimental Results from Harmony VM

that will be analyzed.

The fifth column reports the total number of methods called by this class, computed through a transitive closure, excluding the filtered methods. (This is the number of nodes in the call graph). The sixth column reports the total number of lines of bytecodes (LOBC) analyzed (each instruction in bytecode counted as one line). This number is the actual lines of code being analyzed for all the methods given in the fifth column. The seventh, eighth and ninth columns report the number of *good*, *all_path_secure* (APS) and *risky* methods (a risky method is one that is `public` and not good). These numbers are computed for all the methods analyzed (given in fifth column).

In column nine, the number of risky methods reported are for the entire set of methods that are analyzed, including ones for which summaries have already been computed. For instance, the analysis of `ObjectInputStream` results in no new risky methods, even though the number reported is five. To show this, the number of risky methods that are newly computed in the analysis of the class is shown in column ten. This value for the analysis of `ObjectInputStream` is zero. The eleventh column shows the number of *real risky* methods identified after semi-automatically (i.e., manual analysis with our tool support described later) analyzing the risky methods in column ten with the witness. A real risky method is a risky method that has at least one feasible bad path in practice (not a false alarm).

Summary of results From the results we see that a large fraction of the methods are good. These are methods that are certified to have a security check before any sensitive operation. Overall, in the HotSpot VM, we have found 61 risky methods and in the Harmony VM we have no risky methods.

For instance, a total of 1520 methods were analyzed in the HotSpot VM. 1520 is the cardinality of the set of all methods being analyzed in the fifth column in Figure 5, which resulted in 61 risky methods. Only these 61 methods

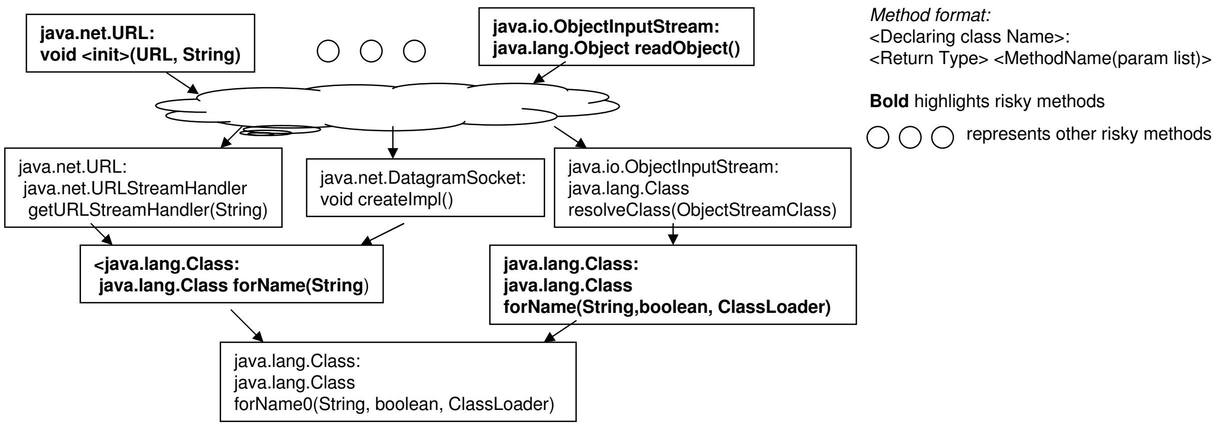
java.net.URL:
void <init>(URL, String)
java.io.ObjectInputStream:
java.lang.Object readObject()
Method format:
<Declaring class Name>:
<Return Type> <MethodName(param list)>
Bold highlights risky methods
represents other risky methods
java.net.URL:
java.net.URLStreamHandler
getURLStreamHandler(String)
java.net.DatagramSocket:
void createImpl()
java.io.ObjectInputStream:
java.lang.Class
resolveClass(ObjectStreamClass)
<java.lang.Class:
java.lang.Class forName(String)
java.lang.Class:
java.lang.Class
forName(String,boolean, ClassLoader)
java.lang.Class:
java.lang.Class
forName0(String, boolean, ClassLoader)

Figure 7: Analysis of risky methods

need to be analyzed further, resulting in a reduction in *two orders of magnitude*. In the Harmony VM, a total of 3928 methods are analyzed, none of these methods was reported as risky.

We consider the reduction in the amount of human effort required to perform this entire effort quite significant. We also provide automation support for manually analyzing the risky methods that we describe below.

Tool support for manual analysis of results Recall that each risky method is a bad method that is `public`. As indicated earlier, corresponding to each bad method we generate a compressed witness. Such a compressed witness for a *bad* method M_0 is a sequence of nodes in the CFG of M_0 ending in a sensitive operation, or ending in node that invokes another method say M_1. In the later case, M_1 itself is a bad method whose witness ends in a sensitive operation within M_1 or ends in call to another bad method M_2. Corresponding to each bad method M_0, we construct a chain of methods $M_0, ..., M_k$ such that there is a path in the expanded control flow graph of M_0 that goes through nodes of all these methods ending in a sensitive operation in M_k. All such chains corresponding to risky methods are arranged as a forest so that the root nodes of the trees in the forest are the bad methods where the sensitive operations are performed without an *a-priori* security check. Such root methods can be automatically identified from the witnesses of the risky methods. These root methods need to be manually analyzed in more detail. This additional step further minimizes the number of methods needed to be manually analyzed.

For the HotSpot VM, all the witnesses of risky methods can be arranged as an (inverted) tree shown in Figure 7. For the sake of space, we show only a portion of the tree close to the root bad methods. This figure shows that all of these 61 risky methods in HotSpot VM are due to a private native method name `forName0` declared in java.lang.Class. This is a method that returns a `Class` object associated with a given class, using the class loader supplied as argument. Since returning the `Class` object can be sensitive, the VM needs to perform a security check.

There are two methods through which `forName0` can be directly accessed. The code of both methods are given in

```
public static Class forName(String className)
      throws ClassNotFoundException {
  return forName0(className,
          true, ClassLoader.getCallerClassLoader());
}
public static Class forName(String name, boolean initialize,
      ClassLoader loader) {
  if (loader == null) {
     SecurityManager sm = System.getSecurityManager();
     if (sm != null) {
         ClassLoader ccl = ClassLoader.getCallerClassLoader();
         if (ccl != null) {
               sm.checkPermission(..);
         }
     }
  }
  return forName0(name, initialize, loader); //native
}
```

Figure 8: forName() code snippet from Hotspot VM

Figure 8.

Both `forName` methods are public and have path(s) from entry node to `forName0` without security check. For example, in the second forName, a path exists when if (load == null) doesn't fall through. The existence of such a path results both methods to be summarized as $\langle inscure\_path, bad\rangle$, and also risky since they are public. The remaining 59 public methods are reported to be risky because they invoke one of the risk `forName` methods directly or indirectly.

After analyzing the code of both `forName` methods manually, we have determined that they are not real risky methods. By passing a null loader to `forName0`, the caller requests the class to be loaded via the bootstrap class loader, which is sensitive. Thus the VM needs to be consulted before loading the class. The absence of a null loader being passed to the class is the case when the VM has already assigned a loader for the class, and therefore there is no requirement for a security check.

After this analysis, we manually updated the $forName$ methods summary from *bad* to not *bad* (i.e., empty bad summary), and our resulting verification run shows that there are zero real risky methods in the HotSpot VM, as shown in column 11 of the results table.
Analysis Time Performance For all the classes tested, the

average time taken by CMV to analyze each class was 74 seconds. The bulk of the time spent is in CFG construction that requires going through methods from several different classes. Ours is a static verification technique, these values are acceptable, also considering the fact that our prototype implementation is currently not optimized for time and space. We are currently exploring an on-the-fly technique that combine CFG construction with the procedure that computes bad and insecure summaries.

Summary In summary, the results suggest that our approach is highly suitable for verification efforts involving large code bases such as the Java standard libraries. These results suggest the approach taken by CMV is scalable and practically useful.

6. CONCLUSION

In this paper, we have presented an approach for checking the complete mediation property for the Java class libraries. Our approach is compositional and is of time complexity linear in the size of the libraries, and hence is scalable for analyzing large libraries, even in the presence of recursive methods. We have implemented this approach in a tool called CMV and used it in checking the complete mediation property for the Java libraries of two widely used JVMs: HotSpot and Harmony. Our experimental results indicate that our approach is scalable and can lead to large reduction in human efforts required for system verification.

7. REFERENCES

[1] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy (SSP)*, May 2002.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *7th International SPIN Workshop on SPIN Model Checking and Software Verification*, London, UK, 2000.

[4] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Computer Aided Verification CAV*, New York-Berlin-Heidelberg, July 2001.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder: Second generation of a Java model checker. In *Post-CAV 2000 Workshop on Advances in Verification*, July 2000.

[6] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM conference on Computer and Communications Security (CCS)*, 2002.

[7] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1986.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. BANDERA: extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[10] U. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.

[11] D. Evans and A. Tywman. Flexible policy directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, California, may 1999.

[12] P. W. L. Fong and R. D. Cameron. Proof linking: Distributed verification of java classfiles in the presence of multiple classloaders. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[13] T. Fraser, J. Nick L. Petroni, and W. A. Arbaugh. Applying flow-sensitive cqual to verify minix authorization check placement. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, New York, NY, USA, 2006.

[14] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *SP'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2006.

[15] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security sensitive operations in legacy code using concept analysis. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, Minnesota, USA, May 2007.

[16] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification CAV*, 2002.

[18] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 1997.

[19] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, 1999.

[20] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, 2002.

[21] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2001.

[22] G. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[23] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer-Aided Verification, CAV '96*, New Brunswick, NJ, 1996.

[24] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995.

[25] J. Saltzer and S. M.D. The protection of information in computer systems. *proceedings of the IEEE*, September 1975.

[26] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[27] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format-string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[28] R. Vallée-Rai and L. H. et al. SOOT - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[29] V.N. Venkatakrishnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *New Security Paradigms Workshop (NSPW)*, 2002.

[30] D. S. Wallach and E. W. Felten. Understanding java stack inspection. In *1998 IEEE Symposium on Security and Privacy*, 1998.

[31] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *USENIX Security Symposium*, 2002.