Trishul: A Policy Enforcement Architecture for Java Virtual Machines

Technical Report IR-CS-45

Srijith K. Nair, Patrick N.D. Simpson, Bruno Crispo, Andrew S. Tanenbaum

Dept. of Computer Science, Vrije Universiteit
1081 HV Amsterdam, The Netherlands
{srijith,psimpson,crispo,ast}@cs.vu.nl

Abstract. The standard Java execution environment provides only primitive support for specifying and enforcing access control policies both at the stack and method call level as well as the higher application level. The current implementation also falls short of providing a secure execution environment for Java applications because of its inability to trace information flow within the environment. In this paper we present the design and implementation of Trishul, a modular information flow control based policy enforcement framework for the Java Virtual Machine. A flexible and powerful language to implement Trishul's policy decision engine is also presented. Performance measurements show that though the prototype implementation does incur overhead, they are within usable limits.

1 Introduction

Since its early days of the 1990s, Java technology has seen wide acceptance in the whole spectrum of computer systems, from backend servers to embedded devices. This was due mainly to the "write once, run anywhere" cross-platform nature of the applications written in Java as well as to the rich programming language available to the application developers.

Much work has been done in making Java secure. Several built-in mechanism like type-safe reference casting, automatic garbage collection and structured memory access makes the language inherently more secure than other commonly used languages like C. Java Virtual Machine's (JVM) features like the class loader architecture and class file verifier further enhance the security of the execution environment.

In the current architecture, the Security Manager (SM) is responsible for managing the access control restrictions (to resources external to the JVM) of the code running inside the JVM.

1.1 Java Security Manager

The Java API asks the SM for permission to perform potentially unsafe actions by invoking the `checkPermission` method. Only if allowed by the SM will the

API go ahead with the execution. If the permission is denied, a security exception is thrown. In Java 2, system administrators can instruct the SM to use predefined policies specified in a *policy file* to make access decisions.

For example, consider an application trying to read the */etc/passwd* file. The Java API would create a `java.io.FilePermission` object and pass the strings '/etc/passwd' and 'read' to the object's constructor. It then passes this `Permission` object to the `checkPermission()` method of the `java.security.AccessController` object. The `AccessController` uses the information contained in the protection domains objects (which encapsulates the permissions granted to the code source in the *policy file*) of classes whose methods are present in the call stack to determine whether the action is to be allowed or not using the stack inspection [1] technique.

1.2 Policy file

A policy file is used to specify `grant permission`(s) to class files loaded into the JVM. In the Java 2 security architecture, each class file is associated with a *code source* which indicates where the code came from. This allows application developers to vouch for codes they develop, using certificates and code-signing.

A permission object has three parts - *type*, *name* and optional *action*. The permission class name indicates the *type*, for example `java.io.FilePermission`. The *name* is obtained from the `Permission` object, */etc/passwd* being an example. The *action* property of the `Permission` object specifies the action requested, for example `read`. One or more such `Permission` objects is associated with a `CodeSource` and forms a `Policy` object. A policy file consists of several such objects as in Listing 1.1.

```
grant signedBy "VU-CA" {
   permission java.io.FilePermission
         "/etc/passwd", "read";
}
grant codeBase
        "file:/usr/share/java/repository/-" {
   permission java.security.AllPermission;
}
```

Listing 1.1. `Policy` object example granting read access to code signed by VU-CA and all access to code in a specific location

1.3 Limitations

Even with all the above-mentioned features, the current SM design still suffers from limitations.

Consider an application that wants `read` access to the `/etc/passwd` file of an UNIX/Linux system. Such an access request is normal, since information present in the file is used to perform several routine housekeeping operations. However, there is no reason why the information obtained from the file should be sent off via the network to an external system. An application which tries to do so could, for example, be trying to harvest user information in order to perform an efficient brute force password attack. What is required is a policy setting that allows an application to read the content of the password file but prevents it from sending that information out on the network.

Policies expressed in the form of currently supported `Policy` objects do not support this level of control. For example, if the policy in Listing 1.1 is used, it will allow read access to the password file but prevent the application from creating a socket connection to a host. But this is too wide a denial since it will also prevent the application from ever sending anything over the network, irrespective of the actual content that the application is trying to send. This is due to the JVM's inability to trace the flow of information in the system and take access control decision at the flow level.

1.4 Contribution

In this paper we present the design and implementation of an information flow tracing based access control architecture to overcome the limitations identified in the Java architecture as enumerated above. Our architecture, consisting of a method call intercepting module, the enforcement decision engine and a language to develope the engine, is generic enough to be used to solve both lower level stack inspection problems as well as the higher application level problem of performing access control based on the flow of sensitive information, as shown in Section 5.

The paper is organized as follows. Section 2 provides a brief overview of information flow tracing. In Section 3 we present an overview of Trishul's architecture while Section 4 introduces the details of the language used to writes policies for Trishul. Example problem scenarios that Trishul helps solve are presented in Section 5 and Section 6 that follows it provides details of Trishul's implementation. The architecture's performance is analyzed and discussed in Section 7. Section 8 looks at some related work in the area, while Section 9 discusses possible optimizations in Trishul, which form part of our future work. We conclude in Section 10.

2 Information Flow Tracing

The power of Trishul is derived from its ability to trace and control the information flow within the JVM execution environment. The theory behind information flow tracing, though old [2, 3], is still an actively researched topic [4, 5]. This section provides a brief overview.

Many programs perform computation by using one or more variables as operands and storing the results in another variable. For example, in the pseudocode $y = x$, when the value of x is transferred to y, information is said to *flow* from object (variable) x to object (variable) y and the flow can denoted as $x \Rightarrow y$ [2]. If the value of x has a security class of $\underline{x}$, the execution of the code makes $\underline{y} = \underline{x}$.

```
boolean x
boolean y
if (x == true)
   y = true
else
    y = false
```

Listing 1.2. Implicit flow code 1

```
boolean b = false
boolean c = false
if (!a)
   c = true
if (!c)
    b = true
```

Listing 1.3. Implicit flow code 2

Flows due to codes like $y = x$ are termed *explicit flows* because the flow occurs due to the explicit transfer of a value from x to y. On the other hand, consider the code shown in Listing 1.2. Even though the value of x is not directly transferred to y, once the code is executed, y would have obtained the value of x. We say that in this case, $x \Rightarrow y$ was an *implicit flow.*

A trickier implicit flow is shown in Listing 1.3. When a is $true$, the first if fails so $\underline{c}$ remains L, where L is the default lowest possible security class in the system. The next if succeeds and $\underline{b} = \underline{c} = L$. Thus, at the end of the run, b attains the value of a, but $\underline{b} \neq \underline{a}$. The same is true when a is $false$. The fundamental problem is that even though the first branch is not taken, the very fact that it is not followed contains information, which is then leaked using the next if.

3 Architecture Overview

In this section, we describe the overall architecture of Trishul while the implementation details are discussed later in Section 6.

3.1 Trishul's Architecture

Trishul's run-time policy enforcement architecture consists of a mechanism to trap the function calls performed by the application and a policy decision engine to check the policy and decide whether or not to allow the calls. The interpreted nature of the bytecode execution within the JVM allows Trishul to interpose itself between the Java application and the lower level system on which it is being executed. In order to provide an application class independent policy enforcement framework, Trishul's design supports a pluggable policy decision engine architecture.

Fig. 1 illustrates Trishul's architecture. It consists of two parts - the core Trishul JVM system and the pluggable policy engine. The core JVM implements information flow tracking and provides the policy engine the hooks needed to trap the calls made by the untrusted application. These hooks allow the policy engine to load appropriate policies into the Trishul system based on the data being used by the application and later, based on the operation being performed on the data by the application, to decide whether or not to allow the application's function call.

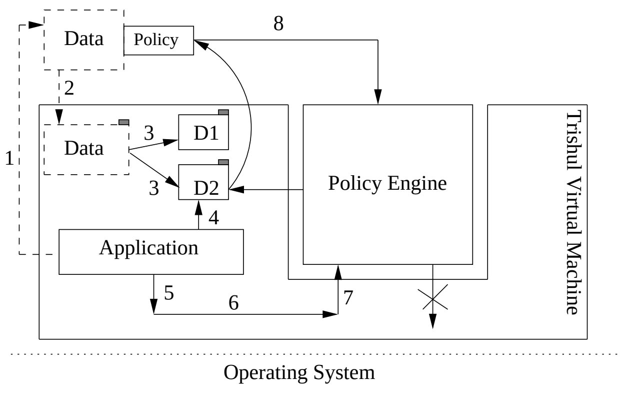

Fig. 1. Trishul Architecture

The system works as follows. When a Java application is started, the application code is loaded and executed in the JVM. A policy engine is also loaded based either on the application being run or via a command-line argument to the JVM. When the application reads a piece of data from say the hard disk (1), it gets loaded into Trishul (2). The policy engine hooks onto the call and attaches a label to the data. The IFC functionality of Trishul ensures that, irrespective of what the application does or where the data is mode in the JVM (3), the label remains associated with the data. When the application tries to act on the data (4), for example send it over a socket connection (5), Trishul interposes (6) and transfers the control to the policy decision engine (7). The engine checks with the respective data's usage policy (8) and decides whether or not to allow the application to proceed. For example, if the application tries to write the data chunk D2, which originated from 'Data' with associated policy 'Do not distribute', to an network socket, the call will be blocked.

As can be seen from the figure, the policy decision engine is a pluggable module separate from the core of the Trishul VM. By allowing the engines to be loaded as pluggable modules, the same framework can be used to enforce policies based on the logic provided by various trusted third parties. For example consider a policy 'play 3 times' associated with a media file. A vendor V_1 would consider a 'play' as having played more than half the file's content, while another vendor V_2 would consider it a play only if the whole file is played fully. Thus V_1 and V_2 would be able to specify different interpretations of the application semantic of 'play' and provide the engine code to enforce the policy.

In addition, Trishul's architecture in itself is not bound to any specific policy language for the data's access policy specification since it is up to the policy engine writer to write the language parser and provide the interpretation logic of policy expressions.

3.2 Handling Implicit Flows

In order to capture the implicit flow of information discussed earlier, Trishul uses the concept of a *context taint*, which extends the idea of associating a security class with the program counter *pc* [6]. It aims to capture the implicit taint labels associated with a code branch, for example a `case` in a `switch` or an `if/else`, by examining the variables that effect the conditional branch and then passing this taint into the branch. Thus, for an `if` control flow instruction (CFI) like `if (a == 5) && (b == 6)` the *context taint* $\underline{ct}$ is computed as $\underline{ct} = \underline{a} \cup \underline{b}$, where $\underline{a}$ denotes the security label associated with the object a. Trishul then tries to identify all objects that are modified within the taken and non-taken branch blocks. When a conditional CFI is actually executed, the objects that are modified in any of the possible paths are tainted with the context taint $\underline{ct}$ using the following rule:

– If the branch is taken: $\underline{object} = \underline{ct} \cup explicit\_flow\_in\_statement$
– If the branch is not taken: $\underline{object} = \underline{object} \cup \underline{ct}$

Let us consider an example using the pseudo-code in Listing 1.3. The analysis at load time computes the $\underline{ct}$ at line 03 ($\underline{ct\_03}$) as $\underline{a}$ and $\underline{ct\_05} = \underline{c}$. Assume $a = false$. Table 1 shows how this approach correctly identifies implicit flow of information from a to b by successfully computing $\underline{b} = \underline{a}$.A similar result is computed when $a = true$.

Line No.	Branch?	Taken?	Taint computation
03	yes	yes	*none* (since branch is taken)
04	no	-	$\underline{c} = \underline{L} \oplus \underline{ct\_03} = \underline{a}$
05	yes	no	$\underline{b} = \underline{b} \oplus \underline{ct\_05} = \underline{b} \oplus \underline{c} = \underline{a}$

Table 1. Branch context taint rule example

To handle implicit flows using context taint, the runtime system must know how the control flow influences the information flow. To determine this, a control flow graph (CFG) is created when a method is analyzed. To actually determine which conditional control flow instructions (CFIs) influences the execution of a basic block, a dataflow analysis is used. Trishul thus employs a hybrid approach by combining the approaches taken by typical *dynamic* and *static* information flow systems by performing a static analysis of the Java bytecode at class load time in order to evaluate and capture as much of the implicit flow as possible and using the taint tracing mechanism at the execution time. Trishul is thus able to make use of the dynamic run-time properties of the system to enforce wider class of policies while at the same time being able to capture implicit flows accurately. Details of the steps involved in this process are skipped here and interested readers are referred to [7].

3.3 Native methods and Exceptions

Native Methods Java applications are allowed to invoke native methods directly using Java Native Interface but since these native methods are no longer run within the JVM and can, among other things, use registers inside the native processor and allocate memory on native stacks, Trishul cannot track the information flow any further. In order to avoid this, Trishul assumes that only trusted native method libraries are allowed to be accessed by Java applications. More on this in Section 6.3.

In order to ensure the integrity of these trusted libraries, the hash of every trusted native library is stored within the JVM as a part of the post-build process. At run-time, these hashes are checked to ensure that the libraries have not been replaced with untrusted ones and only native methods provided by these libraries are then allowed to be invoked by the application.

Exceptions Exceptions can cause unanticipated changes in the control flow, requiring special handling to avoid leaking information. A `throw` statement will transfer control to the appropriate `catch` block. However, the target of a throw statement may not be known before run-time, since an exception that is thrown is just a normal object that resides on the heap, the parameter to the `throw` instruction is a reference to that object. Before run-time, only the static type of the reference can be determined. The actual type of the exception object may be a subclass of that type. As the `catch` block that is invoked depends on the actual type of the exception object, it may not be known before the throw statement is actually executed.

In Trishul, best effort is made to determine the run-time type of the exception object, using a simple analysis that tries to find the instruction that places the reference to the exception on the stack. If this is a new instruction (most often it is), the run-time type is known. In that case, and if there is an appropriate catch block, an edge is added in the CFG from the throw statement to the catch block. In other cases, as a conservative approach, an edge is added to the method's exit block. When the exception is thrown, the current context taint and the exception object's taint are stored and at the catch block this taint is added to the local context taint. [7] provides a more thorough treatment of this subject.

4 Writing Policy Decision Engines

The pluggable policy decision engine handles the actual process of labeling the data when it is introduced into the system as well as making decision on how the data can be used. In order to ease the development of the policy engines, drawing inspiration from the syntax used by the Polymer language [8], a Java-like language named *Trishul-P* was developed. The language provides two main functionality: (1) a mechanism to specify the Java method calls that the policy writer is interested in order for Trishul to transfer the control to the policy engine

```
public Order query (Action a) {
   private policytaint {
      secretTaint , topsecretTaint , pwdFile }
   aswitch(a) {
     case <* java.io.PrintStream.println(..)>:
      return new OKOrder(this, a);
     case <* java.io.PrintStream#<secretTaint>.println
                          (String s#<topsecretTaint>)>:
      return new HaltOrder(this, a);
     case <* java.io.FileInputStream.<init>(..,File f)>:
      if(f.getName().indexOf ("/etc/passwd") >= 0) {
         return new ObjectTaintOrder (a.getThisPointer(),
                                        #object:{pwdFile});
      }
      break;
   }
   return null;
}
```

Listing 1.4. Example of Trishul engine code written in trishul-P

when they are invoked and (2) the steps to be taken when these methods are invoked.

Though the syntax of the policy engine is similar to Polymer, the system works in a different way than Polymer. While Polymer is a Java-to-Java precompiler that rewrites the system libraries and application bytecode permanently, Trishul-P provides hooks for the JVM to intercept method calls and run the relevant engine logic when the application is executed. The actual system libraries as well as application bytecode remains unchanged. The only similarity between the two languages is the syntax.

Trishul-P has three key abstractions: actions, orders and policies. Actions abstract the method calls performed by the Java application. Each time an action specified by the policy writer is about to be executed by an application, the Trishul-P policy is queried for a decision. The decision is returned in the form of a `Order` indicating a specific action the policy engine should take, such as disallowing the action, or allowing it to execute normally.

Let us consider an example engine code fragment in Listing 1.4 for further explanation.

4.1 Actions

The method invocations that the policy engine is interested in intercepting are specified as Action objects represented using the syntax

```
<returnType pkg.class#<thisTaint>.method(..#paramTaint)#<contextTaint>>
```

The constraints that can be specified include the method signature, calling object's name and method's parameters. The policy engine distinguishes between different actions using the `aswitch` statement. This statement is similar to Java's switch statement; the switch expression being an action and the case labels the **action patterns**. Action signatures can also use wildcard patterns: '`*`' matches any one constraint and '`..`' matches zero or more parameter types. For example, the first `case` statement in Listing 1.4 matches the `println` method call defined in the `PrintStream` Java class file of the `java.io` package for zero or more parameters of any type.

4.2 Abstract actions

Consider a policy which is interested in write access to an output channel. Several Java methods can be used to perform this operation and hence a single `Order` would have several actions associated with it. `Abstract` actions makes writing policies in these cases easy by combining several related actions into a single action and referencing this abstract action with the required `Order`. In other words, these `abstract` actions summarizes a set of application method calls into a single action statement.

4.3 Taint Labels & Patterns

A `policytaint` keyword can be used to assign values to taints instead of using numbers like `1` and `2`. Unlike an enum, each name is assigned a different bit in a bitmap, rather than sequential integers. Policy taints can also extend other policy taints, to allow policies to be implemented across different classes.

In addition, taint labels can also be specified as constraints. `thisTaint` specifies the taint of `this` pointer of the class. `paramTaint` can be used to match tainted parameters. It can be specified for individual parameters or '`..`', in which case it matches if any of the parameters is tainted. `contextTaint` can be used to match a tainted context. Thus the second case statement in Listing 1.4 matches `println` method only when the string parameter is tainted with value `topsecretTaint` and when the class instance object is tainted with value `secretTaint`.

Several options are available for matching taints in action patterns. First, when multiple taint patterns are specified, it is possible to match when any taint match occurs, or only when all the taints matches. Second, when matching against an object's taint, either the reference or object taint can be matched. The following pseudo-BNF format is used for taint patterns

```
#<[type:]taint[how][name]>
```

The `type` specifies either `object`, `primitive` or `auto`, to match either an object's taint, a primitive value's taint, or an object taint in case of an object and a primitive taint in case of a value. The main purpose of this is to allow `primitive` to be specified when matching an object, as that allows matching against the reference taint. For example, if a string parameter is matched using `#<object:secretTaint>`, the engine logic will be invoked only if the string object is tainted. `#<primitive:secretTaint>` however, matches when the reference to the string is tainted.

The `taint` is either an integer literal, an asterisk (*) to specify any taint value except 0, or a set of comma-separated taint names enclosed in curly brackets '{}'. `how` is either an ampersand (&) or pipe symbol (|), to match either all bits or any bit. For example, `#<{secretTaint, cryptoTaint} &>` matches if both `secretTaint` and `cryptoTaint` bits are set. On the other hand, `#<{secretTaint, cryptoTaint} |>` matches if either or both of the taint values are set.

The `name` is analogous to a parameter name; it declares a local variable that will receive the taint value that was actually matched.

When tainting or untainting an object, taints can be specified as named literals of the format

```
#[type:]taint
```

The type (`object` or `primitive`) is optional and specifies whether to (un)taint the object or reference. If not specified, object taint is assumed if the taint applies to an object, and primitive is assumed otherwise. Like in taint patterns, `taint` specifies the actual value.

4.4 Orders

Once the policy decision engine intercepts an application's method call that is specified by the actions, it ascertains the consequence of the action and decides on the way to handle the action and returns the decision in the form of a *Order* object. Trishul-P implements the following subclasses of the object:

- `OKOrder`: action is allowed to be executed
- `InsOrder`: decision is deferred until after some specified code is executed and evaluated
- `ReplOrder`: instead of executing action, returns the value specified in the Order
- `SuppressOrder`: suppresses action and throws an exception
- `HaltOrder`: action is not allowed and the application is terminated
- `Param(Un)TaintOrder`: (un)taints a parameter and invokes the action
- `RetVal(Un)TaintOrder`: return value of action is (un)tainted
- `Object(Un)Taintorder`: calling object is (un)tainted
- `ExceptionOrder`: same as SuppressOrder, except that the exception can be specified by the policy
- `CompoundOrder`: allows multiple orders to be combined. This is useful for tainting orders and compound policies

Thus, the third `case` statement in Listing 1.4 causes the policy decision engine to taint the `FileInputStream` object associated with the file `/etc/passwd` with the `pwdFile` taint.

4.5 Combining Policies

Since it is intended that policies may be downloaded or included with data, Trishul allows policy engines to be loaded and unloaded at run-time and as well as combined with the executing engines. Thus, a policy engine is allowed to load new engine classes into the system. These new classes can themselves load other policy engine classes, creating a tree of engine policies. Each engine logic in the tree is allowed to match against any action and return any order. However, a child policy must be at least as restrictive as its parent. As the default action in Trishul-P is to allow an action, this restriction means that a child policy cannot allow anything that its parent forbids explicitly.

This is implemented internally using `CompoundOrder` by placing all the orders returned by the different policies in the `CompoundOrder`, which is then evaluated by the policy engine. In some cases, evaluating the combined orders is straightforward. For example when multiple (un)tainting orders are combined, they are executed in order. However, when a Halt and OKOrder are combined the results must be specified explicitly. In general, when there is a conflict, the most restrictive order is executed. The following rules defines these results. If any order is a `HaltOrder`, the program is halted, while if it is a `Suppress` or `ExceptionOrder`, the first one encountered (in breadth-first search order through the tree of policies) is executed. Any `InsertOrder` is executed for `ReplaceOrder`, the last one encountered is executed. Any of the taint orders are executed when it is encountered.

5 Trishul in action

Now that the Trishul architecture and the policy engine language has been presented, we can provide some examples of how Trishul solves policy enforcement problems that the current JVM cannot.

5.1 Protecting system password file

Consider the problem scenario introduced earlier about access to the '/etc/-passwd' file on UNIX/Linux systems. As explained earlier, an application could have legitimate reasons for reading the content of this file. Since the actual passwords are not stored in plaintext in the file, the read operation by itself is not dangerous. However allowing the application to send the details of the file to the external host via the network would make the system prone to more targeted password cracking attacks since the usernames are known.

In order to prevents this, the enforcement system needs to ensure that the data read from the '/etc/passwd' file is (1) tainted with a label (2) the label is propagated within the system alongside the data and (3) when attempt is made to send the data via the network, it is prevented. With Trishul, this can be done with relative ease. Listing 1.5 shows the fragment of a Trishul-P code used to inform the underlying engine to enforce such an access control.

```
private policytaint {
  pwdF, netC }
aswitch(a) {
case <* java.io.FileInputStream.<init>(..,File f)>:
  if(f.getName().indexOf ("/etc/passwd") >= 0) {
    return new ObjectTaintOrder
     (a.getThisPointer(),#object:{pwdF});
  }
  return null;
case <* *.Socket.getOutputStream(..)>:
  return new RetValTaintOrder(#auto:{netC});
case <* *.PrintStream#<{netC}>.write(..#<{pwdF}>)>:
  return new ExceptionOrder
    (new java.lang.RuntimeException ("Leak!"));
}
return null;
```

Listing 1.5. Policy to protect /etc/passwd

It works as follows. The first `action` and associated `Order` taints the `FileInputStream` object with a label `pwdF` if file being used for initialization is '/etc/passwd' while the second case statement intercepts and returns a network socket labeled `netC`. Trishul's underlying taint propagation mechanism would then ensure that any object that uses this socket would be tainted with the `netC` label while any data read from the `FileInputStream` object would be tainted with `pwdF` label. The third action in the policy file checks for a call to the `write` method of a `PrintStream` object tainted with the `netC` label, which uses `pwdF` tainted data as argument. If the application invokes the method within these taint constrains, it is trying to send the data obtained from the '/etc/passwd' file via a socket to an external host. As a response Trishul returns an `ExceptionOrder` which in turn causes the JVM to throw a `java.lang.RuntimeException` exception.

Couple of notes on the example above. In Listing 1.5, the name of the password file is hard coded to make the presentation easier. In a real implementation, a SELinux-like policy structure would exist for every file that has a usage policy associated with it. The Trishul engine would query the related policy file first and then, based on the policy specified in that file, would perform the engine logic. Attacks like copying the password file to a new file and then reading this new file to perform network actions could be stopped in two ways (1) disabling the copy of the file content to a new file or (2) carrying the policy of the original file to the new file. These are not shown in the example above to keep the code listing simple.

It should also be noted that `PrintStream.write()` is only one of the possible methods that an application can invoke to write data to network. A comprehensive policy engine would use an `abstract action` that encompasses all possible methods that performs similar writes.

5.2 Stack Inspection Limitation

When an application attempts to access a restricted resource, the JVM performs a walk over the execution stack to verify that all the callers currently in the stack has been granted permission to access that resource [1] using the `checkPermission` primitive in order to prevent the Confused Deputy attacks [9]. However it has been shown that the stack-based access control (SBAC) approach [1] is still not secure as it allows untrusted code to influence the execution of trusted code that accesses restricted resources [10]. Consider as example a code fragment, noted in [10], of a trusted library `TrustedLib` in Listing 1.6.

```
public class TrustedLib {
  public static main void(String[] args)
      throws Exception{
    Helper h = new Helper();
    String fname = h.name();
    FileOutputStream f = new
         FileOutputStream(fname);
  }
}
public class Helper {
  public String name(){
    return "/home/user/secret.txt";
  }
}
```

Listing 1.6. Class `TrustedLib` code fragment

Assume that the class `TrustedLib` is provided by a trusted party and is allowed to perform a security sensitive operation like creating a `FileOutputStream`. However, unknown to the user, the class could be using the untrusted `Helper` class to provide it with the name of the `FileOutputStream`, `fname`. When the JVM performs a stack walk, it sees the following callers on the stack - `security.checkPermission`, `FileOutputStream.<init>(File, boolean )`, `FileOutputStream.<init>(String)` and `Trusted.main`.

Since all these callers have permission as strong as `FilePermission "/home/user/secret.txt", "write"`, `checkPermission` will succeed. However this compromises the security of the system because an untrusted code was able to influence the file operation since it was `Helper.name` that provided the file name. Since `h.name` was not in the stack when `checkPermission` performed the stack walk, this influence was not captured by SBAC.

Information-Based Access Control (IBAC) [10] has theoretical been suggested as an approach towards solving the limitation of Java's SBAC model. IBAC works by transforming access-control policy into information-flow policy by associating the permission granted by the former as labels for the latter. When sensitive operations are requested, IBAC systems interposes itself to ensure that code components without the required label is not able to influence the execution of the security-sensitive operation.

Trishul's flexible architecture provides the infrastructure to develop an IBAC prototype. With reference to Listing 1.6, Trishul's prototype implementation of IBAC taints the `String` returned by `Helper.name` which is then propagated as the taint of `fname`. A `checkPermission` action engine logicwould then ensure that all the variables and objects involved in the sensitive operation is untainted (trusted). Thus the taint label attached to `fname` would be caught and in turn trigger an Order that either halts the application or throws an exception.

5.3 Multi-level Security Systems

Multi-level security (MLS) systems take inspiration from defense community's security classifications. Most MLS computer systems use the Bell-LaPadula model [11] that proposes two main mandatory access control security properties. The *no read-up* property states that a subject at a given security level may not read an object at a higher security level, while the *no write-down* property states that a subject at a given security level must not write to any object at a lower security level.

Consider a CEO who has 'Confidential' security clearance. He has two files, one with clearance of Confidential and another with clearance Public, both of which he wants to write into. Current MLS system would require that the CEO open the Confidential file, edit it, close the application and change his current security level to Public by logging out of the system and logging in again with the lower clearance. Only then would he be able to open and edit the Public file. This is needed to prevent the CEO from copying content from the Confidential file and write it into the Public file, which could then be read by someone anyone.

An MLS system implemented within Trishul JVM can avoid the need for the manual change of the current security level, without compromising the security of the system. Trishul achieves this by preventing writes to an object only if this object's level (Public) is lower than the level of the content that is being written (Confidential). Thus the CEO is able to open and edit the Confidential file and the Public file simultaneously and even copy content from the Public file into the Confidential file but will be prevented from copying the data from the Confidential file to the Public file.

Listing 1.7 shows a portion of the Trishul-P policy engine code that was used to prototype such an enhanced MLS system using Trishul. When an application tries to access a protected file on behalf of a subject, the invoked method call (say java.io.FileInputStream) is intercepted and the control is transferred to the policy decision engine. The engine checks the clearance of the subject to access the file and if cleared, it labels the input stream with the security level of the object (specified in a global system configuration file). Trishul then taints any data originating from the input stream with the stream's label and propagates this taint as the data gets used in the system. Later when the application tries to write data into an output channel (OutputStreamWriter), the engine throws a RuntimeException using ExceptionOrder if the output channel's security level label is lower than that of the data being written (confidential > public). Note that in the process above, only the specific instance of FileInputStream is tainted

```
case <* java.io.FileInputStream.<init>(String path,..)>:
   oLabel=objectLevel(path);
   switch(objectLabel) { //confidential=5,public=1...
   case 5:
      return new ObjectTaintOrder (a.getThisPointer(),
                           #object:{confidential},this,a);
....}
case <* *.OutputStreamWriter#<{publiclabel}>.write(
                              ..#<{confidential}>)>:
   return new ExceptionOrder (new java.lang.
                  RuntimeException (``Disallowed"),this,a);
```

Listing 1.7. Enhanced MLS system policy fragment written in Trishul-P

and a new FileInputStream created later will remain untainted, prevent taint spread. While our prototype system uses a custom configuration file t o specify the clearance of the subjects and objects, a production system could use that information provided by a SELinux-like system file.

6 Implementation

Trishul was implemented as a modification of the open source Kaffe JVM [12]. In this section we provide details of some salient features of this implementation.

Kaffe JVM has two operational modes. The interpreter mode executes each Java opcode one at a time, time after time. While the interpreted version is simple to implement and hence extend, its performance is not suitable for running production-quality environment. The issue is addressed by the Just-In-Time (JIT) compiler mode that translates often-used Java opcode into machine code for the system it is running on, providing far better performance than the interpreter, but at the same time makes modifying the JVM a far more complex undertaking. The taint propagation mechanism of Trishul architecture started off as a an interpreter implementation and it was later implemented in the JIT engine too, specifically for the IA-32 CPU architecture.

6.1 Taints

In Trishul, each value that appears in the JVM's stack as well as each local variable, parameter and return value has an associated taint value. The taint values (labels) are stored as bitmaps with the taint combination operator $\cup$ being implemented as the bitwise OR of these values. Each class object has a taint, the *object taint* while each member of the object also has a *member taint*. Whenever an object member is assigned a value, the value's taint is included in the object taint, allowing the object taint to represent the summary of the taints of all of its members, which in turn can be used by the policy writer to evaluate the overall taint of the object. The object taints are stored directly in the memory allocated for the object by the JVM whereas the member taints are stored in specially allocated shadow memory. The reference used to access an object also has a taint value which is included whenever an member is read or written, as is the case with the context taint.

Static variables defined in a class are not object members, but class members. Since there is only a single instance of each static variable in a process, there is only a single taint variable, which is stored in the variable descriptor used by the JVM. Arrays have an *array taint* while each member of the array has an *element taint*. Because arrays are implemented like objects in the JVM (with array elements comparable to object members), these taints are implemented like object and member taints. The `length` variable of an array, which returns an array's length, is not considered to be an object member, since it cannot be modified and is tainted with the array taint.

6.2 Handling Flows

Explicit flows are implemented in a straightforward manner by adding the Java bytecode instructions to combine the taint values when the corresponding values are used in the computation. To handle implicit flows, a two-stage approach is used that is a hybrid between static information flow control systems and dynamic systems, using a static analysis and run-time enforcement. The static analysis of the bytecode, performed at the load time (along with the method verification step), detects the implicit flows while the run-time enforcement allows for policies to be attached when the program is executed, and not earlier in the cycle when it was compiled.

During the load-time analysis, control flow graphs (CFGs) are used to determine the conditional control flow instructions (CFIs), such as `if` statements and looping constructs, that control execution of the current statement. When these instructions are known, the taints from the conditional expressions are also known and can be included in the current context taint. Once the CFG is created, it is analyzed to determine branches in the method's control flow. Whenever the control flow branches, basic blocks that are executed in one path of the branch but not the other(s), must include the conditions' taints in its context taint, as the execution of the basic block, and thus the flow of information in that block, depends on the conditions' result, as discussed in Section 3.2.

When all paths have merged again, the condition no longer influences the flow of control and associated flow of information, so its taint is no longer included in the context taint. The conditions' context taints are called the *partial context taint*; all of the partial context taints in the current execution path together form the *context taint*. Postdominator dataflow analysis is used to detect these branches and merges in flow. The result of this dataflow are summarized in a *context bitmap* which is used at run-time to update the context-taint appropriately.

At run-time, each method's stack frame contains an array of partial context taints, with one entry per conditional CFI in that method. Whenever the conditional CFI is executed, the condition's taint is stored in appropriate entry in the array. Whenever a new basic block is entered, either by an explicit CFI or because the program counter advances outside the current basic block, the context taint is updated. This is done by combining the partial context taints of all conditional CFIs whose bit is set in the new context bitmap.

Each method has an *initial context taint* which combines the context taint of the call site (or 0 for the invocation of the main function) and the taint of the `this` pointer when an object method is invoked. Whenever the context taint is updated, the initial context taint is also included, ensuring that it is not possible to escape a context taint by invoking a different method. When a method call returns, the context taint is updated, restoring it to the value it had before the method call.

6.3 Manual Taint Propagation

While most taints are propagated automatically as described earlier, native method invocations necessiate manual taint propagation. Since these methods also create and move values, they should also be creating and propagating taints. However, since these methods are executed outside the JVM's control, the tainting has to be either implemented as modification to the native code or by using the *annotation* propagation method provided by Trishul. Policy writers can also use this feature for writing flexible policies like, for example, tainting the whole `String` object if one or more characters are tainted.

Annotations are Java classes that contain hook methods for existing classes' methods. When the original method is about to be invoked, control is transferred to the hook method, which has the power to adjust taints before and after invoking the original method. This way, the String example can be handled by writing an annotation for methods that change the array of characters and update the String's taint accordingly.

Annotations can also be defined to specify that some variable does not propagate taint values. This is a clear security violation, but can be necessary in some cases to avoid taint creep. It can be used securely if it is ensured that uses of the variable do not transfer information, as can be the case if the variable is used, say for example, for caching. A similar method is available to allow methods to be invoked without a context taint, again to handle problems certain application scenarios.

In the current implementation, all annotations are stored in a single signed jar file, which is loaded at startup. If its signature is invalid, the system refuses to start, thus preventing normal users and potential unsafe code from adding security compromising annotations into the system.

6.4 Taint Storage

In the JIT mode, Trishul's JIT compiler stores taint values in registers or on the stack as described below.

Register taints Trishul uses the Streaming SIMD Extensions (SSE) registers or the stack to store taint values. IA-32 architecture has 8 128-bit registers (XMM0 - XMM7), each made up of 4 32-bit parts which is mainly used to perform integer operations in parallel on different sets of data. Since Kaffe does not use these registers, Trishul is free to use them.

Each SSE register is used to hold 3 32-bit taints, even though 4 taints would fit. This is due to a limitation of the SSE instruction set: it does not have an operation to move a part of a register into another part of another register, only shuffle operations that combine different parts of a single register into another register. Therefore, moving a part of a register into another register is simulated using 3 operations: the destination part is cleared, the source part is moved into the correct position in a temporary register, and this temporary register is OR-ed into the destination register. As only the destination part must be affected, the remainder of the temporary register must be zeroed. Since the SSE instruction set does not provide an operation to clear a specific part of a register, the highest 32-bit part is always kept to 0, allowing a shuffle operation to copy that 0 into one or more specific parts in a single operation and also move the source part to the correct position. Fig. 2 illustrates the steps needed to move taint of register ECX to taint of ESI.

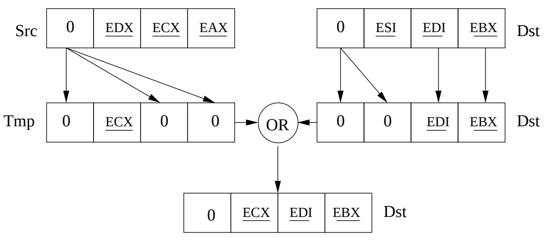

Fig. 2. Moving taint from <u>ECX</u> to <u>ESI</u>

Stack Taints Trishul stores variable and argument taints on the stack. The layout of a typical Kaffe stack is shown in Fig. 3. The arguments to a method are pushed onto the stack before the return address is stored by the call instruction[1].

In the prologue of the newly invoked method, the previous frame pointer is stored and the base pointer register EBP is made to point to the current top of the stack. Below that, the local and temporary variables are stored. Since it is known at compile time how many temporary variables are required, Trishul is able to reference them using addresses relative to EBP, just like local variables. The stack pointer register ESP is used only when a new stack frame must be created.

Fig. 4 shows the layout of a stack frame in Trishul, containing taint values. The taint values for the method's arguments are pushed onto the stack before

[1] Remember that the stack grows downwards

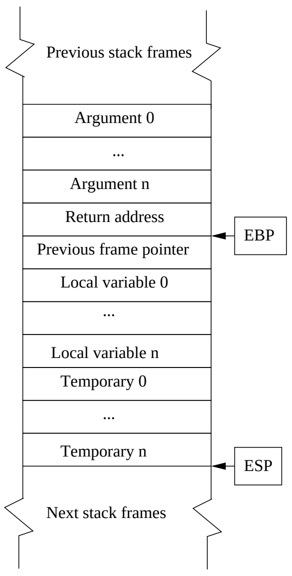

Fig. 3. Kaffe stack frame

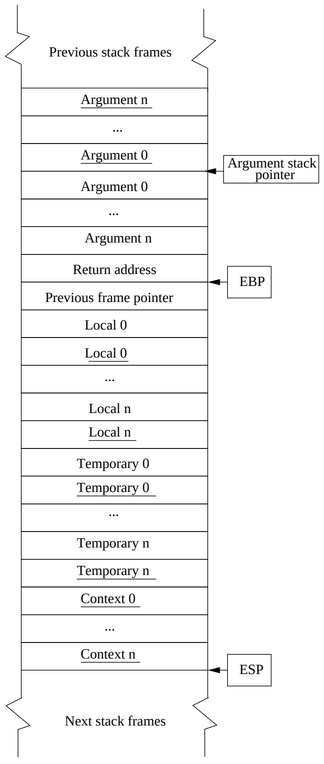

Fig. 4. Trishul's stack frame holding taints

the arguments. This requires that the list of arguments is iterated twice. Had the arguments and taint values been pushed as ⟨argument, taint⟩ tuples, the list would only have to be iterated once, but that would cause problems for native methods, which expect arguments to be pushed using the C's calling convention, which expects them to be pushed one after the other. It would also pose problems for double-sized values, which occupy two consecutive argument slots.

The argument stack pointer (which is stored in a global variable) points to the taint values. This allows native methods to access the taint values. The taint values are pushed onto the stack in the opposite order from the argument values, allowing the taint values to be accessed using the index of the formal parameter as it appears in the C code, whereas the argument values are pushed in the opposite order, as required by C.

Taint values for local and temporary variables are stored as ⟨argument,taint⟩ tuples. Because it is not known how many temporary variables will be required before the full method's code is generated, it is not possible to determine the offset of a taint storage location if the taint values are stored following the variables. This would require two passes of the JIT compiler: one to generate the code and one to fix the offsets of taint values. Using the tuple approach, only one pass is needed, but, as with arguments, double-sized variables present a problem, as they require two consecutive slots. To handle this, the order of variable and taint values are reversed, so that two consecutive slots are used for the value and two for the taint values. Two slots are used for the taint value for a double-sized variable to make accessing slots easier. If a single slot was used instead, finding the proper slot for a variable would require scanning the list of variables to see if any double-sized variables precede the variable.

Context Taints The context taint can be thought of as the taint for EIP register and is stored in a SSE register. The different parts that make up the context taint are stored in the stack frame, as shown in Figure 4. Because the number of local and temporary variables is known at compile time, the offset of the context taint parts (relative to EBP) is known at compile time and can be fixed in the generated code. Entry and exit of basic blocks can be detected when code is generated by comparing the address of the instruction being generated to the addresses in the current basic block. If it is detected that a different basic block is entered, code to update the context taint is emitted.

Exception handling The JIT compiler taints the required objects when an exception is thrown at the time the stack is unwound to locate the exception handler. Tainting these values is complicated by the fact that the code to handle non-taken branches is invoked at JIT-compilation time which generates machine code to handle tainting. The exception code is invoked at run-time, and must taint values directly. To this end, it uses information in the exception's stack trace to locate the run-time information generated by the load-time analysis and the locations of values that must be tainted. When the exception handler is invoked, the runtime simply jumps to the correct address, without an opportunity to initialize the partial context taint array with the exception's taint correctly. Therefore, this is handled when the last stack frame is unwound.

To handle non-thrown exception taint, an unused SSE register is used as a special taint register, allowing easy updating of the taint value. When a method is invoked that may throw an exception, the current value of the taint register is stored on the stack and the register is cleared. When the method returns, the current taint value is stored in the partial context taint array and the original value restored from the stack. As the instruction could throw an exception, it is treated as a conditional control flow instruction and the context taint is rebuilt before the next instruction is executed.

6.5 Trishul-P

The Trishul-P code compiler was implemented using a modified JavaCC [13], while the policy engine itself was implemented within the JVM, allowing it to compare and match static properties of the method call, like the signature, and the dynamic taints of objects, parameters and context taints.

To match against run-time information as efficiently as possible, a two-stage matching strategy is used. During the first stage, which is invoked only when against a specific action for the first time or when the policy changes, the static information is matched. The result of this match is stored and reused whenever the action is executed again. Only when the first-stage match is successful, the second stage is matched in which the dynamic information is compared. This stage must be executed every time the action is executed.

Trishul-P matching is hard in the JIT implementation since the compiler has no access to method and value objects. In order to perform the matching, the object representing that method is required, as it contains the values that must be compared in the Trishul-P match. When a polymorphic method is invoked, this is not known, as only the address of the method's code is known, which is retrieved from an object's dispatch table. To retrieve the method object, the code layout was modified slightly. When Kaffe generates code for a method, it generate a method header (which includes a pointer to the method object), followed by a variable length constant pool, followed by the actual code. This has been rearranged so that the constant pool now follows the code, thus a pointer to the method object is always available at a fixed offset before the code address.

The actual parameters and taint values can be accessed using the argument stack pointer, as described earlier. The return value and taint value can be accessed since the register in which they are stored is known.

7 Performance

In order to measure the overhead introduced by Trishul, its performance was compared against Kaffe. All tests were performed on single node of a four-node AMD Opteron system (model 852, 1Mb cache, 2593 MHz), with 1.5 GB of RAM. All performance measurements were taken using the JIT version in a release configuration and were compared against a standard kaffe-1.1.7 release built using the same compiler options.

The overhead introduced by Trishul architecture can be categorized into three main components - (1) that due to the actual taint propagation mechanism as well as the dynamic calculation of context taints etc. (2) that incurred during the analysis of the bytecode to obtains CFGs, context bitmaps etc. and (3) that introduced by the hooks needed to examine the JVM's method invocations to intercept method of interest to the policy engine. The performance measurements were performed in such a way as to isolate these overheads.

7.1 Taint Propagation Overhead

The run-time overhead due to taint propagation was measured by observing the execution times of the inner loops of a prime number sieve and a file reader program. In order to measure only the runtime of the taint propagation mechanism and not the load-time analysis, the inner loop was executed twice and measured only the second time. The first execution ensures all required classes have already been verified and analyzed. No policy engine is used for the tests to avoid policy engine overhead.

Prime Number Generator A prime number generator was used to test the performance of a CPU-bound application. It loops over the first 16384 integers and determines whether they are prime. As Table 2 shows, an overhead of 167% was observed.

Kaffe	Trishul	Increase
685ms	1828ms	167%

Table 2. Performance of prime number generator

Most of the high overhead can be attributed to the repeated re-calculation of the context taint due to the tight `for` and `if` loops in the algorithm. We are looking at ways to decrease this overhead, as discussed in Section 9.

File Reader This benchmark application measured the performance of I/O-bound applications. The application read a 10Mb, randomly generated, file, into a 64Kb buffer. The data is then printed to standard output, which is redirected to `/dev/null`. As Table 3 shows, a very low overhead in measured for this application.

Since typical real-world applications are likely to be neither fully CPU-bound nor fully I/O-bound, it is expected that the taint-propagation overhead for these applications will be somewhere in between these measures.

Kaffe	Trishul	Increase
7.7ms	7.8ms	1%

Table 3. Performance of file reader

7.2 Load-time overhead

In order to measure the overhead due to the load-time analysis, an application that prints a fixed date (1/1/1970) was used, as this forces a large part of the Java library to be loaded and therefore a large number of analysis to be performed. For example, this specific application run caused 986 methods to be analyzed.

Kaffe	Trishul	Increase
1052ms	1188ms	12.9%

Table 4. Runtime overhead due to load-time analysis

Context	254800 bytes
Non-taken branches	1668928 bytes
Total	1923728 bytes
No. of methods	986
Bytes per method	1951 bytes

Table 5. Memory overhead due to load-time analysis

Table 4 shows that Trishul's load-time analysis incurs a 12.9% overhead comapred to Kaffe. Table 5 shows the memory that is required to transfer information from the load-time analysis to the run-time system. It was measured by recording all allocations of the objects that are used to pass this information; these objects are used exclusively for this purpose. On an average 1951 bytes are required to hold all required information for a single method, the main part being the information on non-taken branches, i.e. the lists of variables that are modified in a branch. Some optimizations that may reduce the size of these lists are discussed in 9.

7.3 Policy Engine Overhead

A microbenchmark application that invoked a specific method 200,000 times repeatedly was used to measure the overhead introduced by Trishul-P's hooks.

Table 6 summarizes the overhead measurements. The first policy (Never matched, static) specified a method that was not invoked by the application at all. It also did not contain any taint comparison and was discarded purely

Policy	Run-time (ms)
None	4.5
Never matched, static	4.6
Never matched, dynamic	31933
Matched, static	212842
Matched, dynamic	212245
Matched, dynamic, order	216000

Table 6. Runtime overhead due to Policy engine

based on static properties of the method's signature. The second policy (never matched, dynamic) specified a method which though was invoked by the application, was not matched due to the specified object taint being different at run-time. While in the first case the overhead was just 2%, the second policy matching process increased the runtime by 7100 times. This big increase is due to the fact that the dynamic properties are checked during the second phase of the two-stage matching process as described in Section 6.5. In other words, the taint value needs to be rechecked every time the method is invoked. Matching on parameter taints and context taints show similar performance results.

The next two policies (Matched, static and Matched, dynamic) matches the method, either the static properties or the dynamic taint values. The larger overhead observed is caused by the work needed to hook into the policy engine: creating objects and arrays expected by the policy engine, installing the security engine, etc.

The last policy (Matched, dynamic, order) also returns a taint order, to capture the overhead of handling an order. When compared to the case with no order is returned (matched, dynamic), this increases the runtime by less than 1%. This shows that hooking into the policy incurs a lot of overhead, regardless of the amount of work done inside the policy engine.

The performance measurement suggests that the most efficient policies are the ones that hook into the policy engine as little as possible, and perform as much work as possible whenever such a hook is eventually made. Note however, that the performance reported here records a worst-case scenario. Such high overhead is not expected of normal applications, since, unlike the microbenchmark application which performs a very tight loop for 200,000 times with only one method being called in the body of the CFI block, they would spend more time calling other methods (not of interest to the policy engine) and performing IO processes in its lifetime, decreasing the overall impact of Trishul-P's hooks.

8 Related work

In order to certify software as complying to a static security policy, Denning proposed a compile-time approach to solve the implicit information flow problem [2]. However the architecture was purely theoretical in nature, depended on the use of specialized 'tagging' supported hardware for supporting tracing and only considered static policies. Newsome and Song [14], among others, used taint tracing to track the use of untrusted data from potentially unsafe input channels, like networks. However, these studies do not consider the enforcement of general usage policies.

Chandra [15] uses a hybrid taint propagation approach similar to Trishul's but by instrumenting the bytecode with propagation code. However the work does not implement any policy expression framework like Trishul-P nor is the architecture flexible enough to implement the range of policies that Trishul can. The work also does not consider the risks posed by native functions nor does it handle exceptions as extensively as in Trishul. Beres and Dalton's hybrid taint propagation approach [5] is limited since it ignores implicit information flows and also relies on enhanced hardware to perform the tracing. JFlow [16] adds security information to Java type system in the form of labeled type. The compiler ensures that no information flow violation occurs by validating the labels when values are assigned to types.

Polymer [8] is a general purpose policy engine for Java that rewrites the application bytecode as well as instruments the system libraries to enforce security policies. While Trishul-P's syntax is inspired by Polymer, the implementation is not only completely independent but also includes the ability to support taint propagation, something which was not considered in Polymer.

9 Future Work

Several possible optimization and fine tuning of the prototype implementation has been identified. For example, currently, the CFGs and context bitmaps of each method used by the application are generated at verification time. This overhead can be reduced by storing the bitmaps and related information of the Java system libraries in a secure, integrity protected manner and reusing it then next time. On a similar line, Trishul creates the CFG separate from the CFG used by the JVM's bytecode verifier due to earlier developmental constrains. However it is feasible to reuse the JVM's internal CFG, thereby decreasing runtime as well as memory overheads.

A reexamination of the way registers are used for storing taints could provide further optimization. Due to the nature of the SSE instruction set, it was observed that accessing individual elements introduced unexpected overheads. Storing a single taint per register could provide an improvement over this. In addition, unused MMX registers can also be used to store the taints. Additional instructions available for manipulating these registers make them more ideal for storing taints.

As of now, the code to handle Trishul-P matches and policy invocations is generated for each method. Every time a method is invoked, it must be checked if the policy has changed, in which case the match must be performed again. It might be more efficient to generate different codes for functions that match the first stage of the Trishul-P match process, thus avoid having to check whether the method matched at each invocation. As a side effect, it would also remove the need to check if the policy has changed whenever a method is invoked, making method invocations a lot quicker. Methods that are not inspected by the policy would incur only the overhead of regenerating the code, which in turn can be reduced by regenerating the code only when the method is used again, which might not happen at all.

The overhead observed in Table 2 can be reduced by not recalculating the context taint in `for` and `while` loops if the CFI's arguments' taints haven't changed within the branch blocks.

10 Conclusion

In this paper we described the design and implementation of Trishul, an information flow control based architectural framework for enforcing policies associated with Java applications. Performance measurements of the prototype shows marked but expected overhead in certain parts of the architecture. Possible approaches to decrease these overheads were identified and form the main focus of future work.

References

1. D.S. Wallace and E.W. Felten, *Understanding Java stack inspection*, Proc. IEEE Symposium on Security and Privacy, 1998.
2. D.E. Denning, *Secure information flow in computer systems*, Ph.D. Thesis, Purdue U., CSD TR 145, 1975.
3. J.A. Goguen and J. Meseguer, *Security policies and security models*, In Proc. of IEEE Symposium on Security and Privacy, Oakland, CA 1982.
4. D. Chandra and M. Franz, *Fine-grained information flow analysis and enforcement in a java virtual machine*, Proc. 23rd Annual Computer Security Applications Conference, Florida, USA, 2007.
5. Y. Beres and C. I, Dalton, *Dynamic label binding at run-time*, Proceedings of the 2003 Workshop on New security paradigms, pp. 39 - 46, 2003.
6. J.S. Fenton, *An abstract computer model demonstrating directional information flow*, University of Cambridge, 1974.
7. S.K. Nair et al., *A virtual machine based information flow control system for policy enforcement*, 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems, Dresden, Germany, 2007.
8. L. Bauer, J. Ligatti and D. Walker, *Composing security policies with polymer*, Proc. ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 305–314, 2005.
9. N. Hardy, *The Confused Deputy: (or why capabilities might have been invented)*, ACM SIGOPS Operating Systems Review, pp. 36–38, 1998.
10. M. Pistoia, A. Banerjee and D.A. Naumann, *Beyond stack inspection: A unified access-control and information-flow security model*, Proc. IEEE Symposium on Security and Privacy, 2007.
11. Matt Bishop, Computer Security: Art and Science, Addison-Wesley, 2002.
12. *Kaffe*, http://www.kaffe.org/, 2007.
13. *JavaCC*, https://javacc.dev.java.net/, 2007.

14. J. Newsome and D.X. Song, *Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation of Exploits on Commodity Software*, 12th Network and Distributed System Security Symposium, 2005.
15. D. Chandra, *Information Flow Analysis and Enforcement in Java Bytecode*, Ph.D. Thesis, University of California, Irvine, 2006.
16. A.C. Myers, et al.*Jif: Java information flow*, http://www.cs.cornell.edu/jif/, Cornell University, 2007.