Design and Implementation of a Virtual Machine Based Information Flow Control System

Srijith K. Nair, Patrick N.D. Simpson, Bruno Crispo and Andrew S. Tanenbaum
Dept. of Computer Science, Vrije Universiteit
1081 HV Amsterdam, The Netherlands
{srijith,psimpson,crispo,ast}@few.vu.nl

Abstract

The ability to enforce usage policies attached to data in a fine grained manner requires that the system be able to trace and control the flow of information within it. This paper presents the design and implementation of such an information flow control system as a Java Virtual Machine, called Trishul. In particular we describe a novel way to address the hard problem of tracing implicit information flow, which had not been resolved by previous systems. We argue that the security benefits offered by Trishul are substantial enough to counter-weigh the performance overhead of the system as shown by our experiments.

1 Introduction

As computers continue to permeate all aspects of life, ensuring the confidentiality of medical, legal, financial, business, and other data is becoming increasingly important. Individual data items may have very specific redistribution policies, such as a medical record intended "only for doctors at this hospital whose patient is John Smith" or "only for employees of the marketing department." Current systems do not do a very good job enforcing detailed and data-specific information flow policies. In this paper we describe the design and implementation of an architecture that can be used to enforce data-specific information flow policies that are more fine grained than what is currently possible.

As an example, consider an e-commerce site that uses an external credit card processing component. The e-commerce application asks the customer for a credit card number and then passes it to the credit card processor over a secure, encrypted channel. However, it wants a guarantee that the credit card processor will not leak it to unauthorized third parties. On the other hand, the credit card processor may be allowed to forward other credit card details (such as name of the card holder) to third party business interests. This is the essence of the problem we are tackling: passing information over to a semi-trusted system and remotely enforcing what can be done with the information.

Our general approach to solving the problem of data-specific policy enforcement at untrusted applications on remote machines is via the implementation of virtual environments whose task it is to track the flow of the data within the remote system. Such a mechanism, often termed *information flow control* (IFC), relies on the ability of the underlying layer to track and control the flow of information without knowing any of the details of what the (untrusted) application is doing or how it works. In this paper we report the implementation of such an IFC system within a virtual machine environment, named *Trishul*.

Obviously, in order to make this design secure, the initiating application needs a way to verify that the virtual environment running on the remote machine is the one it expects and trusts. *Remote attestation* [1, 2] provides a mechanism to enforce this. We will assume that such a mechanism is available and will not

discuss it further. The contribution of this paper is the design and operation of such a virtual environment and the way in which it can control information flow in an application-independent way.

The rest of the paper is organised as follows. We introduce previous works done in formalising the general concepts behind information flow control and discuss in detail the theoretical aspects of such systems in Section 2. In Section 3 we explain the two main approaches commonly used in building IFC systems - language-based static analysis and the less often studied dynamic information flow tracing and control at run-time. We also discuss previous works in using these approaches to build IFC systems and point out their shortcomings.

Our IFC system Trishul is implemented within the context of Java Virtual Machine (JVM). We explain this design choice and introduce the Java architecture in Section 4. In Section 5 we present the details of Trishul's implementation, which was built on the open source Kaffe [3] JVM. In Section 6 we present the results of performance tests conducted using Trishul in order to gauge the overhead added into the system in exchange for enhanced security. In Section 7 we discuss some issues related to the performance, security and usability of the system.

We conclude in Section 8 by summarizing the key points of our work and point towards future work.

2 Information Flow Control

Over the years, access control models like discretionary access control and mandatory access control [4], role-based access control [5] and usage control models [6] have been developed in order to provide a framework for handling the requirement of restricted access to and restricted usage of system resources (like CPU, memory, file system). However, existing work in this area dealt mainly with the theoretical aspects of the models or confine themselves to high level access control implementations that *evaluate* access policies. By itself they can't be used to enforce policies associated with application data.

IFC, by providing a framework to regulate the flow of information within various components of a system, provide a mechanism to implement the actual enforcement mechanism to power the higher level policy evaluation engines.

2.1 Motivating Examples

The work on Trishul was motivated by couple of example applications where enforcement of policies associated with the data input into the system is essential in preserving the security of the data.

The previously mentioned e-commerce example is one such scenario. The e-commerce application can hardly demand that the credit card processor hand over its source code for a security audit. Yet the e-commerce application wants to make sure that confidential data is treated according to its security policy. How can it do this?

Mobile agent systems [7] provide a another scenario. The agents carry with them data whose integrity and confidentiality is important to the security of the system. External nodes that execute the agent code and handle agent data should not be able to use them in a manner that will subvert their usage policy. For example, a piece of data may have an associated policy that its contents should not be stored locally. Enforcing this via a blanket policy of 'no write access to file system' for the whole agent will result in a policy that is too coarse and would very likely prevent the agent from functioning correctly.

Yet another example is that of application that tries to leak user information. Consider an email application which, due to a bug or malicious intent, reads the user's personal files and sends it over to rogue party. Simple access control policies would not be helpful as the user may also like to legitimately email these files to parties he choose. A monitoring system that has an efficient IFC mechanism would be able to trace the usage of these sensitive data, preventing misuse while allowing legitimate use.

2.2 Information Flow Model

Denning formally defined a control flow model FM [8] as

$FM = \langle N, P, SC, \oplus, \rightarrow \rangle$

N denotes a set of storage objects that receive and store data. P is a set of processes that move information around in the system. SC is defined as set of security classes that each object in N is bound to. This security class also includes L, the lower bound of the security classes which is attached to objects in N by default. $\oplus$ is the binary operator that defines the security class of the result of a binary operation performed on any pair of operand classes. $\rightarrow$ denotes the legal *can-flow* of information from one security class of object to another. Without losing generality and specifically in the context of this paper, the objects that form N can be considered as program variables, and in certain instances, as seen later on, blocks of program code.

Many programs compute values using one or more variables as operands and store these values into another variable. For example, in the pseudo-code $y = x$, 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 is denoted as $x \Rightarrow y$.

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

Listing 1: Implicit flow pseudo-code

Flows due to codes like $y = x$ are termed *explicit flows* because the the flow takes place due to the explicit transfer of a value from x to y. On the other hand, consider the code shown in Listing 1. Even though there is no direct transfer of value from x 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*. We discuss the approaches taken by existing information control system to track implicit flows in details in Section 3.4.

3 Managing Information Flow

The conceptual idea behind information flow control and system security based on the model is not new. First explored in the early 70s, it has been researched and used in various forms over the years. In general, two different approaches have been explored - compile time and run time.

3.1 Compile-time Systems

In the *compile-time* approach, programs are written in specially designed programming languages in which special annotations are used by the programmers to specify security labels and constraints to the objects in the program. At compile time, the compiler uses these extra labels to ensure the security of the flow control model. These compile-time checks can thus be viewed as an augmentation of type checking. Continuing with the notations used by Denning [8], $\underline{x}$ *can flow* to $\underline{y}$, denoted by $\underline{x} \rightarrow \underline{y}$, iff information in x is allowed to flow into y. In the context of information flow, the necessary and sufficient condition for a system to be considered secure is that $x \Rightarrow y$ is allowed iff $\underline{x} \rightarrow \underline{y}$.

When information flow occurs between more than two objects, the compiler has to verify that each of the flows is allowed. For example, in the code segment $z = x + y$, it is clear that information flows from both x and y to z. A compiler would, in theory, need to verify $\underline{x} \rightarrow \underline{z}$ and $\underline{y} \rightarrow \underline{z}$. In general, if $b = f(a_1, a_2....a_n)$,

each $\underline{a_i} \rightarrow \underline{b}$ has to be verified. However for the sake of simplicity, the compiler computes $\underline{A} = \underline{a_1} \oplus \underline{a_2}...\underline{a_n}$ and then verifies $\underline{A} \rightarrow \underline{b}$. Readers are referred to work by Denning and Denning [9] for a detailed analysis of the model, including various code structures, assignments, control and data structures and procedure calls.

Compile-time information flow analysis was used by Denning [8, 9] as a mechanism aimed at adding a certification mechanism into the compiler analysis phase in order to prove the security of the system. In JFlow [10], an example of a modern compile-time system, the Java [11] programming language is extended in order to let the programmer specify security labels to the objects. At compile time, a special compiler uses the labels to verify the information security model of the system. Once this has been verified, the code is translated to normal Java code and a normal Java compiler transforms it into bytecode.

3.2 Run-time Solutions

While the compile-time approach assigns labels to each object and uses them to make sure information flow occurs only if the security labels allow it to, the run-time solutions take a different approach. They use the concept of labels as extra 'properties' of the object and track their propagation as the objects are involved in computation. Instead of verifying $\underline{x} \oplus \underline{y} \rightarrow \underline{z}$ at compile time, the system propagates the security class of the information source into the information receiving object. Thus, the assignment $\underline{z} = \underline{x} \oplus \underline{y}$ occurs.

These assignments however only track the flow of information as it moves through the system. The actual enforcement of security policies is carried out by another part of the system, hereby termed the 'policy engine'. It intercepts all information flows from program objects (such as variables) to output channels, and allows the flow to proceed only if they are not disallowed by the relevant policies. Examples of such output channels are files, shared memories, network writes etc. Whenever an object x tries to write information into an output channel O, the policy engine checks whether $\underline{x} \rightarrow \underline{O}$ is allowed by the specified policy and if not, the flow is disallowed. The implied assumption is that the policies refer to (or can be translated to) restrictions based on the the usage of these restricted channels.

Fenton's Data Mark Machine [12] is one of the earliest systems that used the concept of run-time information flow control to enforce policies. However the machine was an abstract concept and no implementation was ever attempted. The security mechanism proposed by Gat and Saal [13] works in a similar fashion. The system however relies heavily on specialized hardware architecture to trace information flow. The RIFLE architecture [14] is a more recent system that implements run-time information flow security with the aim of providing policy decision choice to the end user. They use a combination of program binary translation and a hardware architecture modified specifically to aid information flow tracking. Again, the use of the modified hardware architecture prevents it from being used on a normal machine.

Beres and Dalton [15] use the DynamoRIO [16] dynamic instruction stream modification framework to dynamically rewrite machine code in order to support dynamic label binding. The underlying concept behind the architecture of our system *Trishul* resembles that of this system with an important practical difference: instead of using a separate code modification framework, we make use of interpreted nature of Java's bytecode instructions to perform dynamic tracing as explained later on. TaintBochs [17] uses a similar idea to track flow of information within a system but with the aim of tracking how 'tainted' data flows in the system. With a similar objective in mind Haldar et al. [18] use bytecode instrumentation to track tainted data received from the network. They also attempt to extend this idea by using bytecode instrumentation to perform mandatory access control on Java objects, in order to enforce security policies [19]. However, the level of granularity that is considered, objects, is too coarse grained to be useful in many applications. For instance, they provide as an example a class method that tries to leak a secret file into a public file [19]. This is prevented by tagging the whole class instance as 'secret' as soon as the secret file is read and denying access to public channels once this tag has been set. The coarse nature of this tagging however prevents the class method from accessing any public channels even if the operation it wishes to perform is not on the data read from the secret file.

3.3 Comparison

Recent years have seen considerable interest in research of the compile-time approach towards information flow. A recent survey [20] references around 140 papers on language based security. One of the reasons for favoring the compile-time approach is the belief that these systems leak only one bit of information per program execution and hence are inherently more secure than run time systems [10]. However, it has been shown by Vacharajani et al. in [14] that termination channel attacks, usually considered the Achilles' heel of run-time systems, can be engineered to leak the same amount of information in compile-time systems as in run-time ones.

Compile-time systems suffer from the important limitation that the policies are bound to the code in a static manner. There is no easy way to handle scenarios where different policies need to be attached to different runs of the application using different input data. In a similar way, these system perform policy-code binding early in the life-cycle, preventing their use in application scenarios where the policy is bound, not to the application but instead, to the data. An example where such limitations occur is that of an email system in which each incoming email has its own distribution policy, none of which are constant across application runs or known at compile time. Compile-time systems fail here.

Compile-time systems are in general more efficient than run-time in that the verification is done only once, at compile time. At run-time, these systems can thus confine themselves to checking the proof of the verification. However, run-time systems performs flow control on each run of the code, slowing the system. The gain in speed enjoyed by compile-time systems however is in exchange for the limitation on the kind of policies that can be enforced. These include policies that depend on the dynamic run-time properties of the system and the user. For example, a policy that states 'This application should not be allowed to send more than 1 MB of data across the network in one day' cannot be verified at compile-time, since the enforcement requires the maintenance of a state that tracks the network usage of the application at run-time. Similarly, compile-time systems cannot ensure the enforcement of system-wide obligations [22] that may be stated in the usage policy, unless they can be expressed at compile-time in a static, immutable manner.

Compile-time systems are written in special languages; hence most existing applications, written in C, C++, or Java, will have to be rewritten in these languages before they can be verified. Yet another shortcoming is that the verification process is performed by the programmer and the user has to trust the programmer. Although proof carrying codes [21] can be used to enhance the trust, practical use of the concept hasn't reached a critical mass.

Inline reference monitors (IRMs) [23] use an hybrid reference monitor with post-compile time (but not strictly run-time) code rewriting approach to the problem of high-level policy enforcement. However, Schneider has shown that information flow, not being a *safety property* is not enforceable by the use of reference monitors [24]. Hence, because they are unable to trace information flow within the system, in order to enforce a fine grained policy like 'do not allow data accessed from /secret to be sent over the network,' IRMs have to resort to enforcing a coarser policy like 'do not allow data accessed from anywhere within the local file system to be sent over the network.'

3.4 Handling Implicit Flows

From the earlier discussion in Section 2.2, it should be clear that implicit information flows are harder to track and verify than the explicit ones. Over the years, several approaches have been suggested that provide varying degree of security against information leaks from implicit flows while trying to build a system that is not overly restrictive.

The Data Mark Machine can be modified to handle implicit information flow by adding a new security class for the program counter $\underline{p}$. Whenever a control branch occurs, $\underline{p}$ is set to the $\oplus$ of the class of objects that form the arguments of the branch decision. Within the branch block, $\underline{p}$ is added to every control flow.

Thus in the example illustrated in Listing 1, when the if statement is executed, $\underline{p}$ is set to $\underline{x}$ and $\underline{y}$ is set to $L \oplus \underline{p} = \underline{x}$. Thus the implicit information flow from x to y is captured by the security label $\underline{y}$ and the process $\underline{x} \rightarrow \underline{p} \rightarrow \underline{y}$.

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

Listing 2: Implicit flow pseudo-code 2

However a different implicit flow example, first expressed by Fenton [25] and shown here in Listing 2, shows that the Data Mark system is still not fully secure. When a is $true$, the first if fails so $\underline{c}$ remains L. The next if succeeds and $\underline{b} = \underline{p} = \underline{c} = L$. 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 underlying problem is that even though the first branch is not followed, the very fact that it is not followed contains information, which is then leaked using the next if.

A trivial (and ineffective) approach to this problem is to ignore it, as done by Beres and Dalton [15]. Fenton [26] and Gat and Saal [13] proposed a solution which works by restoring the value and class of objects changed within the branch structure, back to the *value* and security class it had before entering the branch. This however would not work in practice since existing application codes routinely use similar control structures without paying any consideration to information flow leaks.

Aries [27] takes a more drastic approach wherein a *write* to an object within a branch structure is disallowed if its security class is less than or equal to the security class of the program counter, $\underline{p}$. Thus, in the previous example if a is $false$, when the program tries to write to c, the compile time system prevents it from doing so, since c's security class $L <= \underline{p}$ (= $\underline{a}$). This approach works only if the security classes have an explicit notion of high and low. Furthermore, $\underline{p}$ may not be known during compile-time.

Denning [28] proposes a more secure approach whereby the compiler inserts an extra instruction at the end of the $if(!a)\{c = true\}$ code block to update $\underline{c}$ to $\underline{p}$ (= $\underline{a}$). Thus, irrespective of whether the branch was followed or not, the class of object acted upon within the branch is updated to reflect the information flow. Our implementation takes inspiration from this approach, but rather than perform the required class update at compile time we perform them at Java method invocation as explained later in Section 5.2.

4 IFC and Java architecture

Having discussed previous approaches and their limitations, it is now time to explain our proposal.

Run-time information flow control systems can be implemented at several different abstract levels within a computer architecture. Implementing it at the application level, ties down the system to a specific application. Implementing it within the operating system, as in HiStar [29], allows the operating system to enforce various kinds of access restrictions by controlling information flow between kernel objects, like threads, address space and devices. However, such an implementation is not able to enforce application level usage policies. For example, if an email application is allowed read access to mail files and write access to network devices (the normal case), it becomes very difficult to enforce a selective 'do not forward' policy for some emails but not others.

Another aspect to be considered is that the process of tracing information flow at run-time involves having to dynamically trace access to stacks, registers, program counter and memory region. Current CPU architectures do not, however, provide this flexibility, forcing run-time systems proposed to date, such as RIFLE [14] and dynamic label binding [15], to rely on the use of 'enhanced' hardware.

With these design considerations in mind, application virtual machines stands out as an obvious middleware platform choice for implementing Trishul. Originally introduced as a means to provide an abstract machine architecture for application developers to write their code without bothering about the underlying machine architecture, application virtual machines abstract away the machine architecture, preventing programs from directly accessing the platform registers and physical memory. Furthermore, the interpreted nature of the architecture makes it particularly suitable for implementing run-time flow analysis. As the Java virtual machine (JVM) is one of the most widely deployed virtual machine environment around, it was chosen for implementing *Trishul*.

4.1 Java Architecture

The Java architecture comprises two distinct environments: compile-time and run-time. In the compile-time environment, programs written in Java programming language are compiled into machine architecture independent *bytecodes* using the Java compiler and stored in what are called *class files*. At run-time, an abstract computer called Java Virtual Machine (JVM) loads these class files and executes them in a platform-dependent manner.

The simplest implementation of the JVM is an *interpreter*, which executes each bytecode instruction one at a time. While interpreters are easy to implement, they take longer to execute the programs than *just-in-time compilation* implementations, which compile portions of the bytecode into native machine code. This approach however is much more difficult to implement and could potentially introduce security holes in the flow tracing mechanism. Due to these complications and the resource constrains of our developmental effort, we chose to implement Trishul in the interpreted mode by modifying the Kaffe interpreter engine to keep track of the information flow.

The Java architecture provides a level of built-in policy-based security. The early implementation used the concept of *sandboxing* to create two levels of security environment. This was refined later on (JDK 1.2 and above) to provide more levels of security environments whose security permission could be specified with a finer granularity [31]. Cryptographic signatures are used to bind the application code to the origin of the code and policies are defined based on the principals (origin) of the code. However, the supported policies cannot be expressed with enough granularity to be of use in IFC-based enforcement. For example, users could trust applications signed by the principal of *sun.com* and give it permission to read local files and create connections to all *.sun.com* domains. However, as this policy could only be applied at the level of the overall application, it is not possible to enforce application-semantic level policies such as ‘do not allow the application to write data originating from /secret to any network connection’.

4.2 JVM Internals

The JVM specifications [30] define the functionality that every virtual machine implementation should support, while leaving design choices to the individual developers. This has helped develop several JVM implementations. We modified the open source Kaffe [3] JVM to implement Trishul. In this section we describe the internal design of the JVM that is relevant to the implementation of Trishul. A detailed treatment of the full design aspects of JVM is beyond the scope of this paper and interested readers are referred to other resources [32].

An interpreter JVM has three distinct parts (1) the class loader which is responsible for loading classes and interfaces and performing associated security checks; (2) the execution engine which executes each bytecode instruction; and (3) the runtime data area. The runtime data area consists of a method area, heap, Java stacks, native method stacks and a program counter (pc) register. Each Java application is run inside a separate virtual machine. The method area and the heap are shared across all threads running in a JVM.

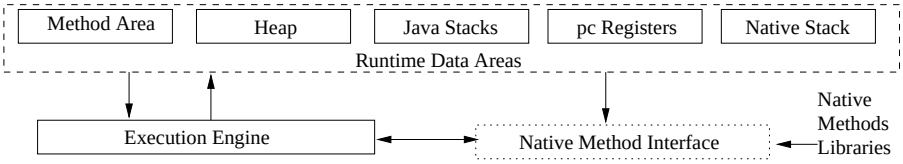

Figure 1: Internal layout of a JVM

The method area holds per-class structures including method data, method code and constant pool, while the heap holds all the objects dynamically instantiated by the VM.

The Java architecture consists of two kinds of methods – native and Java. Java methods are written in the Java programming language, compiled into bytecode, stored in classes, and interpreted by the JVM. Native methods are typically written in C or C++ and compiled into machine code and stored as architecture specific system libraries. They usually provide direct access to host resources. Java code can call native methods directly from the JVM using the Java Native Interface (JNI). Direct access to these native methods however renders the calling code platform specific, so their use is discouraged. Instead, a Java distribution is packaged with a set of Java classes that abstract away the native method calls.

Every thread started in the JVM is given a separate Java stack that is used to maintain the state of all Java methods called by the thread, like the local variables, intermediate calculations and parameters used for its invocation. The state of the native methods invoked by the thread is saved using separate native method stack, registers as well as platform specific memory areas. If the thread is executing a Java method, the program counter (pc) register indicates the next instruction to be executed. This internal layout of the JVM is represented in Figure 1.

Each Java stack is made up of frames, with each frame containing the state of a separate Java method invocation. In interpreter mode, Kaffe JVM uses the variables array to hold the local variable values and the operand stack to hold intermediate operation results.

Throughout the rest of the paper we use *'taint'* to represent that a variable has a specific security class value and that, an operation that involves a tainted variable/slot *propagates* the taint.

The VM executes the instructions by moving data from the local variable array to the operand stack or vice versa and performing computation on these values in the operand stack using it also to store intermediate values. In order for the virtual machine to track the flow of information as the instructions are executed, every slot on the variable array as well as the operand stack has to be extended to store the label of the information that is stored in the slot. The next section describes the actual implementation details of Trishul.

5 Implementing Trishul

In this section we describe in detail our implementation of Trishul, a run-time IFC JVM system based on version 1.1.7 of the Kaffe JVM.

5.1 Stack, Heap and Object Taints

As explained in the previous section, in order to implement taint labels on the local variables and temporary values, the stack structure has to be extended. In Kaffe each stack is implemented as a set of slots. Trishul extends the C struct that implements the slot to hold the taint information, as shown in Appendix A.1.

As in the original slot implementation, the memory allocation is handled automatically by Kaffe's stack management functions. However, taint propagation has to be instrumented separately since Kaffe copies members of the stack slots rather than the structures as a whole. The taint propagation mechanism was added by extending the macros used by Kaffe to implement the Java instruction set. Appendix A.1 shows

the modification to one such macro, move_int(), used by several instruction sets like ILOAD to move integer value between variables and operand stack.

In addition to the stack, Java object members and array elements can also be tainted and hence they too need to be extended to hold the taint labels. In Trishul, these taint labels are stored on the heap, in one of two ways. Static Java object member taint labels are stored in the structure used by the virtual machine internally to handle the field and its values. Non-static object members and array elements [1] are stored in shadow memory allocated when the object (or array) is allocated. This memory is released by a modified garbage-collection function that provides a reference to the shadow memory to the garbage collector, which in turn uses it to deallocate the memory automatically.

A pointer to the shadow memory, called member_taint, is added to the structure that represents a basic Java object. The structure representing an object field was then extended to include an index into the shadow memory. This identifies the location in the shadow memory where the taint for that field is stored. In the case of static members, the index is reused to store the actual taint value. When an object member is written, the taint_object_store macro is invoked. This macro updates the member_taint array for the field being written. Likewise, when a member is read, the taint_object_load macro is invoked to read the values. Appendix A.2 lists the relevant code changes.

5.2 Branch Context Taints

As discussed in Section 3.4, implicit information flows are difficult to trace. In order to handle them in a correct manner, we propose the concept of a *branch context taint*, which extends the idea of associating a security class with the pc. It aims to first 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 context 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 $\underline{ct} = \underline{a} \oplus \underline{b}$. Please note here that while the examples in the paper are presented using pseudo-codes and Java codes, Trishul actually works at Java bytecode level.

In order to capture the implicit information flow that is available even when a branch is not taken, we try to identify all variables that are modified within the branch blocks. To this end, a list of variables that are modified in each block is calculated at class load time and stored with each basic block, as shown in the rightmost column in Figure 2. When a conditional CFI, like a branch or goto, is executed, only one possible path is taken. The variables that are modified in any of the other paths, but not in the current path (since they are tainted by execution of the path anyway), are tainted with the context taint $\underline{ct}$ using the following rule:

- If the branch is taken: $\underline{object} = \underline{ct\_of\_object} \oplus explicit\_flow\_in\_statement$
- If the branch is not taken: $\underline{object} = \underline{object} \oplus \underline{ct\_of\_object}$

Let us consider an example using the pseudo-code in Listing 2. 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 summarises the actions taken at run-time by the IFC system.

We see that this approach correctly identifies implicit flow of information from a to b by successfully computing $\underline{b} = \underline{a}$. A similar (correct) result is computed when $a = true$.

Trishul uses a two-stage process to handle context taints. In the first stage, when a method is invoked for the first time, its control-flow graphs (CFGs) with *branch bitmaps* are computed to detect context blocks. In the second stage, these CFGs and branch bitmaps are summarised into *context bitmaps*. These processes are explained in details below.

[1] After all an array is also an object.

Line number	Is it a branch	Is branch taken?	Taint computation
03	yes	yes	$none$ (since branch is taken)
04	no	-	$\underline{c} = \underline{L} \oplus ct\_03 = \underline{a}$
05	yes	no	$\underline{b} = \underline{b} \oplus ct\_05 = \underline{b} \oplus \underline{c} = \underline{a}$

Table 1: Branch context taint rule example

5.2.1 Creating the CFGs

A CFG is created using a single forward pass over the method's code with a node for each basic block. A basic block is a sequence of instructions with a single entry-point (the first instruction) and a single point of exit (the last instruction). A CFI always forms the last instruction of a basic block. Directed edges represent transitions between basic blocks, either caused by the normal flow of instructions or by a CFI.

Note that a basic block may have an outward edge leading to a special *exit* block for the last instruction in a method or one other block for a goto or block without CFI, or two other blocks for if-statements, or any number of other blocks for switch instructions. CFIs that exit the current method (return and throw instructions) are linked to the *exit* block ensuring that all blocks (other than the exit block) will have at least one outward edge.

To ensure that each basic block has a single point of entry, the CFI's targets are checked. If the target is before the current program counter (i.e. a backward branch) and it branches into the middle of a basic block, the basic block is split so that the target instruction is the starting point of its block. In the case of a forward branch, a new basic block is created starting at the target instruction, which is initially empty. This block is stored in a forward list, which is checked when a new basic block is created. Later when the basic block that includes the target instruction (identified earlier in the forward branch) needs to be created, the basic block from the forward list is used. If the target instruction is not the first instruction of the new basic block, this block is split as required.

```
public static void main(String args[])
{
    boolean a = true;
    boolean b;
    if (a)
    {
        b = true;
    }
    else
    {
        b = false;
    }
}
```

Listing 3: Java code for CFG example

```
00: iconst_1
01: istore_1
02: iload_1
03: ifeq 11
06: iconst_1
07: istore_2
08: goto 13
11: iconst_0
12: istore_2
13: return
```

Listing 4: Bytecode of Listing 3

5.2.2 Branch bitmaps

A branch bitmap is associated with each basic block. This contains a number of bits for each conditional CFI, one bit for each possible target of the CFI. In the case of an if-statement there are two bits: one representing the case when the branch is taken, and one representing the case when the branch is not taken. A switch instruction has one bit per case, and possibly one bit for the default case.

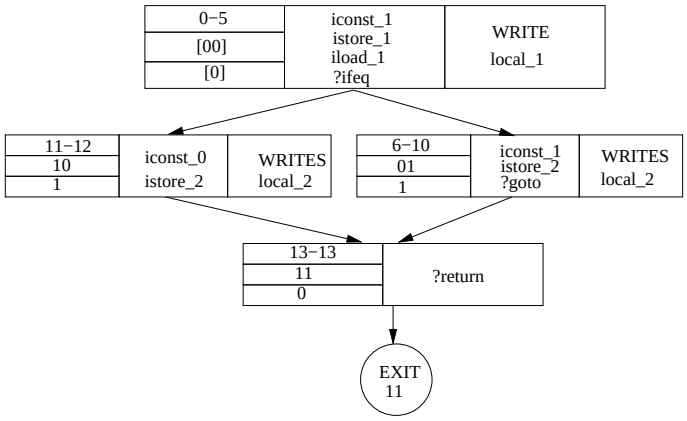

Figure 2: Control-flow graph created from Listing 4

The branch bitmaps are shown in Figure 2 in the center-left field in each node. The CFG in Figure 2 is created from the bytecode in Listing 4, which in turn was compiled from the Java code in Listing 3. In this case, the bitmap consists of two bits, both referring to the if-statement in the topmost basic block. The fact that these bits represent the if-statement at the end of this basic block is indicated by the rectangular brackets that enclose these bits.

Initially the branch bitmaps are initialized to zero. Bits that represent a branch target are initialized to one in the basic block containing that instruction. In other words, the bitmap in basic block <11 12> (indicating the program counters in the top-left field) is initialized to 10, because block <0 5> branches into this block. Likewise, block <6 10> is initialized to 01. Block <13 13> is initialized to 00[1], as the earlier branch instruction does not branch directly into it.

Once all bitmaps have been initialized, they are continuously updated until each bitmap satisfies the condition that each bit that is set in any block that precedes the block in question is also set in the current bitmap. In other words, each bit that is set in a block flows into every block following it.

Once this is done, the bits controlled by a specific CFI can be in one of two states: all bits have the same value, or they have different values. In the first case, each possible path starting at the CFI includes the basic block, or no path includes the basic block. Either way, the execution of the basic block is not controlled by the CFI. When the bits have different values, only some of the paths starting at the CFI reach the basic block, therefore the execution of the block is controlled by the CFI.

5.2.3 Context bitmaps

Context bitmaps summarize the information stored in branch bitmaps. The bitmap contains a single bit per CFI. The bit is set if the basic block is controlled by the CFI represented by that bit. Context bitmaps are shown in Figure 2 in the bottom-left fields. Again, rectangular brackets are used to show which bit represents the CFI in a basic block. The basic blocks <6 10> and <11 12> are controlled by the ifeq instruction in block <0 5>, while block <13 13> is not.

The context bitmaps are used at run-time to detect which context taints must be enabled. They are currently stored in a list, sorted on the program counter of the first instruction in the block. When a CFI is executed, or the program counter runs out of the current block, the new context bitmap is retrieved from the list.

[1]The figure shows a value that is attained at the end of the analysis.

5.2.4 Context taints

When a conditional CFI is executed, the combined taint of all values used in the instructions expression(s) is stored in the context taint array. This array contains an entry per conditional CFI, and thus has as many entries as the context bitmap has bits. When a new basic block is entered, its context bitmap indicates which conditional CFIs control the execution of the block. As the context taint includes the taint of any expression that controls execution of the current basic block, the taints stored in the context taint array must be combined for all conditional CFIs whose bits are set in the context bitmap.

5.3 Tainter Class

When policy-tagged data is used by an application that runs within Trishul, it is tainted with a security label. In order to implement this, Trishul adds hooks in order to intercept calls from the application to core Java methods that import data from input channels into the system. This is done by the *Tainter* class.

When the application tries to perform an output channel operation using the tainted data, like writing to the local file system or the network, Trishul again intercepts these calls to ensure that the application is allowed to perform the operations as per the policy associated with the data. If the policy forbids this usage, an exception is thrown. The list of method invocations that triggers the taint and the exceptions are specified via a policy file that is passed to the JVM as a commandline argument.

6 Performance

The ability to track and control information flow within the system comes at the expense of performance overhead. This overhead is introduced at two distinct points. In order to implement an effective policy based IFC system, the Tainter class has to examine every access to possible input channels in order to decide if the data is to be tainted, as per the policy. Similar hooks are present to examine access to output channels too. This overhead can vary drastically based on the granularity of the policy specified. For example, if a policy states that any file read from the path '/secret' has to be tainted, the hooks needs to examine each invocation of the 'FileInputStream' class constructor to check for the path name. However, if the policy states that any file with the string "Secret Information" in its content has to be tainted, the hooks need to examine each invocation of methods that can read data from the file, like the 'DataInputStream.readLine' function, which could be called and intercepted as many times as there are lines in the file.

Once the hook introduces the taint, the actual taint propagation mechanism introduces the other overhead. This can be attributed to, among other things, the analysis of the CFGs, the calculation of the context taints and the creation and maintenance of the taint properties of the objects.

We performed some benchmark measurements to evaluate the amount of overhead introduced by Trishul. The experiments were conducted on an Intel Pentium M processor 1.60GHz machine with 512MB RAM, running Ubuntu 6.10 with a 2.6.17-10-generic SMP Linux kernel. As mentioned before, Trishul was implemented on version 1.1.7 of the Kaffe JVM, and was compared to the same [1].

IBM's jMocha microbenchmark suite [33] provides a set of performance tests designed to measure the performance of operating system services of JVM implementations. Table 2 summarises the result of the 'AllObjectConstruct (large assign)' benchmark which records the time taken to construct objects and initialise all local variables. The three values are for varying number of initialisations. The test reflects the overhead introduced mainly by the creation of the CFGs and the creation and initialisation of the taint labels to their default values. Though an overhead of 29% seems big, since this benchmark measures only the

[1] Compiled using config: ./configure –disable-gtk-peer –with-staticlib –with-staticbin –with-staticvm –with-engine=intrp –disable-vmdebug CFLAGS=-O3

	1	2	3
Kaffe	3.53	6.09	8.95
Trishul	4.56	7.89	11.55
% overhead	29.18	29.56	29.05

Table 2: jMocha benchmark, AllObjectConstruct (large assign) in μs

	256	512	1K	2K	4K	8K	16K	32K
Kaffe	47.57	74.86	113.68	135.77	147.95	170.39	190.08	193.37
Trishul	44.29	71.78	109.56	130.9	145.42	168.46	189.22	192.7
% overhead	6.89	4.11	3.62	3.59	1.71	1.13	0.45	0.35

Table 3: jMocha benchmark, FileWriteBW in MB/s for various block sizes

initialisation time, which forms a very small part of the full runtime, we feel that the observed overhead is acceptable.

Table 3 compares the bandwidth (in MB/sec) of writing to a file while Table 4 compares the bandwidth attained in reading from a file, both of 16M size, for both Kaffe and Trishul JVMs. Note that in both cases, as with the AllObjectConstruct benchmark, no taints were introduced into the Trishul system.

	256	512	1K	2K	4K	8K	16K	32K
Kaffe	68.33	116.48	154.5	208.19	273.99	343.24	386.52	419.01
Trishul	65.05	111.77	148.25	201.06	271.05	340.48	386.38	418.6
% overhead	4.8	4.04	4.05	3.42	1.07	0.8	0.04	0.1

Table 4: jMocha benchmark, FileReadBW in MB/s for various block sizes

The jMocha file operation benchmark results show that the maximum overhead introduced by Trishul is 7% which reduces to a very reasonable value of 0.4% for large block sizes. This variation can be explained by the observation that when the files are read/written in smaller block sizes, the loop that performs the read/write, is executed more times and each time Trishul has to calculate the new branch taint at each CFI instance. These, being expensive operations, introduce more overhead into the system.

Next we considered a program that opens a file, reads the content one line at a time and writes the entire content into another file [1]. We introduced taint into Trishul by setting the policy 'taint data obtained from any file with the string $secret$ in the file's path'. The time taken for the application to work on files of various sizes was measured. Each line of the file contained 32 characters. Table 5 summarises the time taken (in ms), averaged over five runs.

The results show that compared to Kaffe, Trishul introduces an overhead of around 25% in execution time. While this is not a negligible overhead, we feel that this penalty is a reasonable price to pay for the additional security offered by the flow control and policy enforcement functionality we obtain in exchange. Furthermore, we are confident that the overhead can be reduced by further optimisations.

Next the performance of Trishul and Kaffe when running a brute-force prime number generator was compared. In such an application the time spent in propagating the taint label across stacks as well as that spent in executing branch context related calculations overshadows the time needed to create, initialise and destroy taints. It provides one of the worst-case overhead scenarios in using Trishul and any similar IFC system. The application goes through the first N integers to check if it is a prime number or not. In Trishul, we tainted the integer under consideration in two ways. In one case we associated a taint label with the

[1] The Java code is provided in Appendix

File size	128 KB	256 KB	512 KB	1024 KB	2048 KB
Kaffe	1962	3890	7714	15403	30668
Trishul	2450	4831	9627	19153	38186
Overhead %	24.8	24.2	24.7	24.3	24.5

Table 5: Time taken to read a file's content and write it into another file

Number of integers	16384	32768	65536
Kaffe	3987	14148	52741
Trishul (no taint)	4835 (21.3)	17814 (25.9)	66402 (25.9)
Trishul (taint introduced once)	9548 (139.4)	35270 (149.2)	131457 (149.2)
Trishul (taint introduced individually)	9583 (140.3)	35363 (149.9)	132373 (150.9)

Table 6: Time in ms taken to check for prime numbers. Numbers in bracket is the % overhead.

variable used in the integer generation loop once, which is then propagated as the variable is used within the application. In the other case, we tainted each integer separately after it was generated. The performance of these two scenarios were compared against a pure Kaffe implementation and a Trishul implementation with no tainted data.

Table 6 summarises the execution time, in milliseconds and averaged over five execution runs, needed in each case for different number of integers considered. The numbers in brackets indicate the percentage overhead compared to the Kaffe's performance.

As the results show, the computational overhead incurred by Trishul with no tainted data is much less compared to that incurred when taints are introduced. The very high overhead reported is indicative of the large amount of context taint related computations triggered by the tight loops in the code.

7 Discussion

Performance optimisation

Trishul is still in the early stages of development and some of the performance numbers obtained in the previous section can probably be improved by further optimisations. Overhead incurred during the creation of the CFGs and similar action performed at application class load time cannot be eliminated as they have performed once in the class's life-cycle irrespective of the taint of the data under consideration. However, the CFGs of trusted core Java libraries used in Kaffe and Trishul, like GNU Classpath [34], could be calculated in advance and stored in a secure manner and be re-used each time they are needed.

Reducing the overhead due to context taint calculations could potentially improve performance further. For example, an analysis of the branch blocks could reveal that the objects used in context taint calculations are never written into and hence their security class never change within the branch block. This information can then be used to skip the repeated context taint calculations performed when backward branch loops are encountered.

Policies

The Trishul architecture proposed here does not use a specific built-in security policy model, allowing it to be used as a policy-model independent, generic IFC system. Different policy models can thus be used as long as the policies can be translated to the level that Trishul can operate on. We are in the process of developing an example policy model and associated translator. We are also working on implementing

index	policy
5 = 00010001	display: no
6 = 00010010	network: no
7	.....

Figure 3: Policy table

a secure data source policy-tagging system along the lines of file and network labeling implementation of multilevel security systems [35].

Whatever be the policy model, its representation within Trishul influences the efficiency and security of the IFC system. Consider the code shown in Listing 5:

```
String a = ``No display''
String b = ``No network''
String c = a + b;
```

Listing 5: Policy bitmap pseudocode

If a has been assigned the security policy "do not display" and b was assigned "do not send over the network," c formed by the concatenation of a and b should inherit both the security policies and have an effective policy "do not display and do not send over the network" ($\underline{c} = \underline{a} \oplus \underline{b}$). In order for an IFC system to support such compound policies, the policies have to be represented internally in an efficient format. We plan to represent policies in Trishul by bitmaps of a configurable size. For example, the labels could be implemented as a n * k bitmap lookup to a table that stores all the policies used by the virtual machine instance. Thus, when the VM starts, it would initialise a table to hold all the policies needed by the system. Every time some policy-bound data is introduced into the system, the system creates a new entry in this table. The object that stores the data is then given a label that points to the index of the corresponding policy in the table. The policy table is constructed within the VM memory and is not available to the application for reading or modifying.

For the example in Listing 5, when the JVM initialises a and reads its policy, assuming that it is a new policy, the JVM creates a new entry into the table as shown in Fig. 3 it creates a new entry in the table at index 5. $\underline{b}$ is given a value 6. Using a 8-bit bit array in a 4x4 format, 5 can be expressed as '00010001' and 6 as '00010010'. Thus $\underline{c} = \underline{a} \oplus \underline{b} = 00010001 \oplus 00010010 = 00010011$.

Native methods

As mentioned earlier in Section 4.2, Java applications are able to invoke native methods directly using the JNI. Once invoked, the 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. Hence there is no way for the IFC system to track the information flow once these methods are invoked. In order to avoid this, applications have to be prevented from passing tainted data as arguments to the native methods. Since native method invocations are performed by the JVM, such selective disabling is possible.

8 Conclusions and Future Work

In this paper we described the design and implementation of Trishul, a JVM based information flow control system. Using the concept of branch context taint, explained in the paper, Trishul is able to tackle the problems associated with implicit information flow tracing.

Performance measurements using Trishul show that the system incurs limited overhead. Optimisation needed to reduce this, some of which have been outlined in the paper, form part of our future work. We also

plan to use Trishul as the basis for the development of an actual policy enforcement architecture that can understand and enforce policies expressed at application semantic level.

References

[1] Trusted Computing Group, *Trusted Platform Module Main Specification, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands, March 2006, Version 1.2, Revision 94*, http://www.trustedcomputinggroup.org.

[2] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, *Design and Implementation of a TCG-based Integrity Measurement Architecture*, 13th Usenix Security Symposium, San Diego, California, August 2004.

[3] *Kaffe.org*, http://www.kaffe.org/, 2007.

[4] *Trusted Computer Security Evaluation Criteria*, DOD 5200.28-STD, Department of Defense, United States of America, 1985.

[5] D.F. Ferraiolo and D.R. Kuhn, *Role Based Access Control*, 15th National Computer Security Conference, 1992.

[6] J. Park and R. Sandhu, *Towards usage control models: beyond traditional access control*, Proceedings of the seventh ACM symposium on Access control models and technologies, pp. 57–64, 2002.

[7] G. P. Picco, *Mobile Agents: An Introduction*, Journal of Microprocessors and Microsystems, Vol.25 (2), pp. 65–74, 2001.

[8] D. E. Denning, *A lattice model of secure information flow*, Communications of the ACM, Vol.19 (5), pp. 236–243, 1976.

[9] D. E. Denning and P. J. Denning, *Certification of programs for secure information flow*, Communications of the ACM, Vol.20(7), pp. 504–513, 1977.

[10] A. C. Myers, *JFlow: Practical Mostly-Static Information Flow Control*, Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL), pp. 228–241, San Antonio, Texas, 1999.

[11] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, ISBN 0-201-63451-1, 1996.

[12] J.S. Fenton, *An abstract computer model demonstrating directional information flow*, University of Cambridge, 1974.

[13] I. Gat and H.J. Saal, *Memoryless execution: a programmer's viewpoint*, IBM Tech. Rep. 025, IBM Israeli Scientific Center, 1975.

[14] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani and D. I. August, *RIFLE: An Architectural Framework for User-Centric Information-Flow Security*, Proceedings of the 37th International Symposium on Microarchitecture, 2004.

[15] Y. Beres, C. I, Dalton, *Dynamic label binding at run-time*, Proceedings of the 2003 Workshop on New security paradigms, pp. 39 - 46, 2003.

[16] *DynamoRIO System Overview*, http://www.cag.lcs.mit.edu/dynamorio/doc/system.html

[17] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher and M. Rosenblum, *Understanding Data Lifetime via Whole System Simulation*, Proc. 13th Usenix Security Symposium, 2004.

[18] V. Haldar, D. Chandra and M. Franz, *Dynamic Taint Propagation for Java*, Proc. Annual Computer Security Applications Conference, 2005.

[19] V. Haldar, D. Chandra and M. Franz, *Practical, Dynamic Information Flow for Virtual Machines*, 2nd International Workshop on Programming Language Interference and Dependence, 2005.

[20] A. Sabelfeld, and A.C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications, Vol. 21(1), pp. 5–19, 2003.

[21] G.C. Necula, *Proof-carying code*, Proc. 24th ACM Symposium on Principles of Programming Languages, pp. 106–119 1997.

[22] M. Hilty, D. Basin and A. Pretschner, *On Obligations*, Proc. 10th European Symposium on Research in Computer Security, LNCS, 3679, Springer-Verlag, 2005.

[23] U. Erlingsson, *The inlined reference monitor approach to security policy enforcement*, Ph.D. Thesis, Cornell U., 2004.

[24] F.B. Schneider, *Enforcable security properties*, ACM Transactions on Information and Systems Security, Vol. 3(1), pp. 30–50, 2000.

[25] J.S. Fenton, *Memoryless subsystem*, Computer Journal, Vol 17(2), pp. 143–147, 1974.

[26] J.S. Fenton, *Information protection systems*, PhD. Thesis, University of Cambridge, 1973.

[27] J. Brown and T.F. King Jr, *A Minimal Trusted Computing Base for Dynamically Ensuring Secure Information Flow*, Tech. Report ARIES-TM-015, MIT, 2001.

[28] D.E. Denning, *Secure information flow in computer systems*, Ph.D. Thesis, Purdue U., CSD TR 145, 1975.

[29] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres, *Making information flow explicit in HiStar*, Proc. 7th Symposium on Operating Systems Design and Implementation, pp. 263–278, 2006.

[30] F. Yelling and T. Lindholm, *The Java Virtual Machine Specification*, Addison-Wesley, 1996.

[31] L. Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation. The Java Series*, Addison-Wesley, Reading, MA, USA, 1999.

[32] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill Companies, ISBN 0071350934, 2000.

[33] E. Gluzberg and S. Fink, *An Evaluation of Java System Services with Microbenchmarks*, IBM Research Report RC 21715 2/3/2000, 2000.

[34] Free Software Foundation, *GNU Classpath*, http://www.gnu.org/software/classpath/, 2007.

[35] Hewlett-Packard Co., *H-UX 10.16 CMW Security Feature Guide*, 1996.

Appendix A Taint Implementation in Trishul

A.1 Stack and Heap Taints

```
typedef struct _slots{
  union {
    jint     tint;
    jword    tword;
    jlong    tlong;
    jfloat   tfloat;
    jdouble  tdouble;
    void*    taddr;
    char*    tstr;
  } v;

} slots;
```

Listing 6: Old slot struct

```
typedef struct _slots{
  union {
    jint     tint;
    jword    tword;
    jlong    tlong;
    jfloat   tfloat;
    jdouble  tdouble;
    void*    taddr;
    char*    tstr;
  } v;
  taint_t taint; /*unsigned int*/
} slots;
```

Listing 7: Modified slot struct

```
#define move_int(t, f) (t)[0].v.tint = (f)[0].v.tint /* old */
#define move_int(t, f) (taint1(t,f), (t)[0].v.tint = (f)[0].v.tint)
#define taint1(t,f1)   (t)[0].taint = (f1)[0].taint
```

Listing 8: Example of a macro modification to support taint propagation

A.2 Object Taints

```
typedef struct Hjava_lang_Object{
  struct _dispatchTable* vtable;
  struct _iLock* lock;
} Hjava_lang_Object;
```

Listing 9: Old object struct

```
typedef struct Hjava_lang_Object{
  struct _dispatchTable* vtable;
  struct _iLock*  lock;
    taint_t *member_taint;
} Hjava_lang_Object;
```

Listing 10: Modified object struct

```
typedef struct _jfieldID{
  Hjava_lang_Class* clazz;

} fields;
```

Listing 11: Old object field

```
typedef struct _jfieldID{
  Hjava_lang_Class* clazz;
  (...)
  union
  {
    /*For static fields*/
    taint_t taint;
    /*For object fields*/
    int taint_index;
  } trishul;
} fields;
```

Listing 12: Modified object field

```
#define taint_object_store(obj,field,f)
taintAdd3 (FIELD_TAINT((obj)->v.taddr,field),
        objectTaint(obj), (obj)->taint, (f)->taint)
```

Listing 13: Macro that updates member_taint array

Appendix B Performance measurement codes

B.1 Read from and write to file

```
package trishul.test;

import java.io.*;

class SecretRead
{
   static void main (String args[])
   {
      try
      {
         long start_time = System.currentTimeMillis();
         FileInputStream in     = new FileInputStream (args[0]);
         DataInputStream reader = new DataInputStream (in);
         StringBuffer contents = new StringBuffer();

         while (reader.available() !=0) {
            String str= reader.readLine();
            contents.append(str);
            contents.append(System.getProperty("line.separator"));
         }

         String s = contents.toString();
         Writer output = null;
         File aFile = new File("blah.txt");
         output = new BufferedWriter( new FileWriter(aFile) );
         output.write( contents.toString() );

         long end_time = System.currentTimeMillis();
         System.err.println("Time taken = " + (end_time - start_time) + "ms");
      }
      catch (Exception e)
      {
         e.printStackTrace ();
      }
   }
}
```

B.2 Prime Number Generator

```
package trishul.test.taint.arith;

import java.io.*;

class ArithPrimeTimeTaken
{
   private static boolean isPrime(long i)
   {
      for(int test = 2; test < i; test++)
      {
         if(i%test == 0)
         {
            return false;
         }
      }
      return true;
   }

   public static void main(String[] args) throws IOException
   {
      int n_loops =  32*1024;
      int n_primes = 0;

      int j;

      long start_time = System.currentTimeMillis();

      //j = taint(0,0x03); //Taint once
      j=0;

      for(; j < n_loops; j++)
      {
         int i = j;
         //int i = taint(j,0x02); //Taint every value

         if(isPrime(i))
         {
            n_primes++;
         }
      }

      long end_time = System.currentTimeMillis();
      System.out.println(n_primes+" primes found in "+(end_time - start_time)+"ms ");
        }
}
```