A Virtual Machine Based Information Flow Control System for Policy Enforcement

Srijith K. Nair[a], Patrick N.D. Simpson[a], Bruno Crispo[a,b] and Andrew S. Tanenbaum[a]

[a] *Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands*

[b] *DTI, University of Trento, Italy*

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, named Trishul, as a Java Virtual Machine. In particular we address the hard problem of tracing implicit information flow, which had not been resolved by previous run-time systems and the intricacies added on by the Java architecture. 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.

Keywords: Informaton Flow, Run-time Monitoring, Policy Enforcement

1 Introduction

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 email infrastructure in a corporate environment where the email sender wishes to enforce email policies like 'do not forward' without having to trust email application running on the recipient's system. Mobile agent systems provide another scenario. The agents carry with them data whose integrity and confidentiality is important to the security of the system. External nodes that

[1] This work has been supported by NWO project ACCOUNT 612.060.319 and partially by the EU project S3MS IST-STREP-27004.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

execute the agent code and handle agent data should not be able to use them in a manner that will subvert their usage policy.

Our general approach to solving the problem is via the implementation of a virtual environments whose task it is to track the flow of the data within the remote system and enable the enforcement of policy decisions. Such a mechanism, often termed *information flow control* (IFC), relies on the ability of the layer to track and control the flow of information without knowing any of the details of the (untrusted) application's inner working or actions. In this paper we report the design and implementation of such an IFC system within a virtual machine environment, named *Trishul*. In order to make this design secure, the initiating entity needs a way to verify that the virtual environment running on the remote machine is the one it expects and trusts. We believe that Remote Attestation [1] provides such a mechanism and will assume that such a mechanism is available and will not discuss it further. We also do not deal with the issue of information leaks through certain covert/side channels attacks like resource usage and timing attacks in our work.

The rest of the paper is organised as follows. In Section 2, after discussing the approaches various approaches towards implementing a secure flow tracing system, we present our proposal for handling information flows. We present the details of Trishul's implementation in Section 3. In Section 4 we present the results of performance measurement tests conducted using Trishul. In Section 5 we discuss some issues related to the performance, security and usability of the system and in 6 we examine related work in this field. We conclude in Section 7 by summarizing the key points of our work and point towards future work.

2 Information Flows

2.1 Explicit and Implicit Flows

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 can denoted as $x \Rightarrow y$ [2].

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

Listing 1: Implicit flow code 1

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

Listing 2: Implicit flow code 2

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*. In order to capture this implicit flow of information, Trishul uses the concept of a *context taint*, which extends the idea of associating a security class with the program counter *pc* [3]. 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 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 $\underline{ct} = \underline{a} \oplus \underline{b}$, where $\underline{a}$ denotes the security label associated with the object a.

An even more tricky implicit flow is shown in Listing 2. When a is $true$, the first if fails so $\underline{c}$ remains L, where L is the lowest possible security class in the system. The next if succeeds and $\underline{b} = \underline{pc} = \underline{c} = L$. 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.

In order to capture these classes of implicit information flows, we try to identify all objects that are modified within the branch blocks. To this end, a list of the objects that are modified in each block is calculated at class load time and stored with each basic block. When a conditional CFI is 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: $\underline{object} = \underline{ct} \oplus explicit\_flow\_in\_statement$
- If the branch is not taken: $\underline{object} = \underline{object} \oplus \underline{ct}$

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.

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

We see that 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 $a = true$.

2.2 Managing Information Flow

In general, two different approaches have been explored with the aim of providing information flow control - compile time and run time.

In the *compile-time* approach, applications are written in specially designed programming languages in which special annotations are used to attach 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. $\underline{x}$ *can flow* to $\underline{y}$, denoted by $\underline{x} \rightarrow \underline{y}$, iff information in x is allowed to flow into y [2]. In the context of information flow, the necessary and sufficient condition for a system to be considered secure is that, for all (x, y), $x \Rightarrow y$ is allowed iff $\underline{x} \rightarrow \underline{y}$ [4].

Run-time solutions take a different approach by using the labels as extra property of the object and tracking 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.

Compile-time system suffer from the limitation on the kind of policies than can be enforced by it. Since the policies are bound to the code in a static manner early in the life-cycle, these systems cannot be used in application scenarios where the policy is bound, not to the application, but instead to the data. These systems also cannot enforce policies that depend on the dynamic run-time properties of the system. Pure run-time systems on the other hand are not able to handle implicit flow leaks due to their inability to analyse flows associated with unexecuted branches, as shown in Listing 2. Trishul uses a hybrid approach, as explained later on, whereby a static analysis of the Java bytecode is performed at class load time in order to evaluate and capture as much of the implicit flow as possible. Later on, when the code is executed, the actual flow of taints is traced. Trishul is thus able to make use of the dynamic run-time properties of the system to enforce wider class of policies.

3 Implementing Trishul

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 [5], allows the operating system to enforce control information flow between kernel objects, like threads, address space and devices. However, such an implementation will find it very difficult to translate application level usage policies into OS system calls due to the semantic gap between the two levels. 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.

With these design considerations in mind, application virtual machines stands out as an obvious middleware platform choice for implementing Trishul. The interpreted nature of these systems make it particularly suitable for implementing run-time flow analysis. Java virtual machine (JVM), being one of the most widely deployed virtual machine environment around, was chosen for implementing Trishul.

When a piece of data, with a policy attached to it, 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 Java library methods that brings data into the JVM. In the complete architecture of Trishul, a pluggable policy engine module specifies which of these methods are of interest to the engine and hence need to be intercepted. Similarly, the engine also specifies which output channels are protected and which of the associated method calls need

to be intercepted. A comprehensive treatment of the actual implementation of this pluggable policy engine module and the language for writing the policy engine is beyond the scope of this paper and is merely mentioned in Section 5. Instead, this paper focuses on the implementation of the core information tracing mechanism of Trishul based on version 1.1.7 of the Kaffe JVM.

3.1 Java Architecture

The Java Virtual Machine (JVM) is an abstract computer which loads the class files produced by the Java compiler and executes the bytecode contained in them in a platform-dependent manner. Even though Java architecture provides a level of built-in policy-based security [6], the supported policies cannot be expressed at the granularity or semantics we are interested in.

Trishul was initially implemented in the interpreted mode of the Kaffe JVM [7] wherein each bytecode instruction is executed one at a time, as a proof of concept. The alternate just-in-time mode, which provides much better performance, is currently in the process of being implemented.

An interpreted 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 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. 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.

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.

3.2 Stack, Heap and Object Taints

In order to implement taint labels on the objects, the stack structure has to be extended. In Kaffe each stack is implemented as a set of slots. Trishul extends the struct that implements the slot in order to hold the taint information. 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 as Kaffe copies members of 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. In addition to the stack, Java object members and array elements also need to be be tainted. Therefore they are extended to include a taint label for each member variable.

To handle implicit flows, the 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, as in Figure 1. After the first basic block, the control flow branches into two different paths, representing the if/else statement in the source code in Listing 3, whose bytecode representation is shown in Listing 4. In the branch that is executed, the context taint must include the taints on the condition used in the if-statement. After the branches merge, at *pc* 13, the if statement no longer influences the flow of control, so the condition's taint should no longer be included in the context taint.

This information is specified in the *context bitmap* (shown in the bottom left section in the CFG). Each basic block has such a bitmap, which contains a single bit per conditional CFI. If the bit is set, execution of the basic block is influenced by that CFI, so its condition's taint must be included in the context taint whenever the basic block is executed. To this end, an array of partial context taints is maintained, which has an element for each conditional CFI. Whenever the condition is evaluated, the appropriate element in the array is updated with the condition's taint. When a new basic block is executed, the context taint is reconstructed by including only those elements of the partial taint array whose bit is set in the bitmap.

```
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

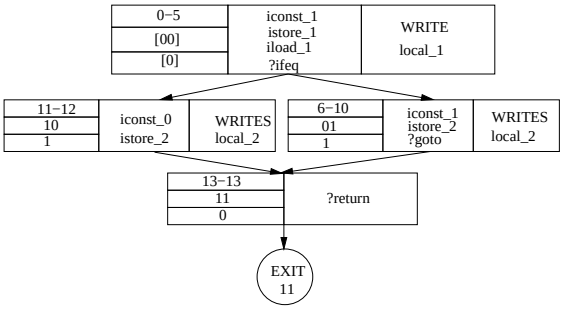

Fig. 1: Control-flow graph created from Listing 4

To actually determine which conditional CFIs influences the execution of a basic block, a dataflow analysis is used. Whenever there is a branch in the CFG, each outgoing path is tagged with a different value. After the dataflow analysis, the first node where all the tags are set, is the node where all execution paths merge. In the CFG, the center left section shows the tags for this analysis. In the context bitmap, the bit for the condition CFI is set in the basic blocks where the paths have branched, but not not yet merged.

3.4 Fallback mechanism

In some cases it cannot be determined which variables are modified in a branch before the branch is actually executed, either because it depends on information that is only available at run-time or because the analysis is not rigorous enough. To ensure that no information is leaked, we use a conservative fallback mechanism, a *global taint*, $\underline{gt}$. This global taint is always used in computing the context taint ($\underline{ct} = \underline{ct} \oplus \underline{gt}$). Whenever the variables modified within a branch cannot be determined, the branch's context taint is included in global taint, ensuring that any later uses of the variables in the non-taken branch will be tainted with the branch's context taint. Of course, as the global context taint is not untainted reset, it must be untainted manually to avoid label creep [8]. In Trishul, the policy engine is tasked with explicitly untainting the global taint.

Trishul is able to track all accesses to local variables, temporaries and stack variables, and in a lot of cases object members. The system cannot track object members accesses if the object's 'this' pointer cannot be determined at the time of analysis. We are actively considering how to improve our load-time analysis to reduce the number of times the fallback mechanism is invoked and thereby the label creep. A way to do this would be to contain the scope of the fallback mechanism, for example by limiting it to a method's block. This would be safe if it can be determined that all values that should be tainted will be overwritten before the method returns. We are also considering to allow for annotations to optionally disable the fallback mechanism within Trishul. This could help avoid label creep in cases where our analysis fails but no information is actually being leaked. Of course, these annotations must be used very carefully to avoid information leaks, and therefore they can only be specified within the trusted policy engine.

3.5 Exception handling

As exceptions can cause changes in the control flow, they require special handling to avoid leaking information. In Java there are two kinds of exceptions: normal exceptions and run-time exceptions. A normal exceptions must be handled explicitly, either by containing the offending throw instruction in an appropriate 'catch' block, or by declaring it as part of the method's signature. Run-time exceptions are used to signal internal errors that may not be recoverable, such as dereferencing a null pointer or division by zero. Exception handling in the JVM is identical for normal and run-time exceptions, but the fact that run-time exceptions are not declared makes their analysis harder, as does the fact that most instructions can cause a run-time exception. Because these instructions are so common and the likelihood of a run-time exception is very low, tainted run-time exceptions may be treated as an abnormal event, causing the program to terminate. In this case, they can be disregarded in the exception analysis.

A *throw* statement will transfer control to the appropriate catch block. This is like a *goto* statement, with the difference that the target address may be in a different method if the throw statement is not inside an appropriate try/catch block. Also, unlike a *goto* statement which always has a fixed target address, 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, the catch block may not be known before the throw statement is actually executed. Also note that if the exception is thrown to a different method, it is generally not possible to determine the catch block before run-time, as that would require knowledge of each possible call site.

In our approach, 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 (which it frequently is), the run-time type is known. If the run-time type is known, and there is an appropriate catch block, an edge is added in the CFG from the throw statement to the catch block. In other cases, an edge is added to the method's exit block. This errs on the side of caution as it assumes no catch block will be executed, the variables written in any of the catch blocks will be tainted, possibly triggering the fallback mechanism.

Method invocations also require special care in the light of exceptions. If a method can throw an exception, the flow of control will not necessarily pass to the instruction following the method invocation, but may instead pass to a catch block or the caller of the method. This turns a method into a conditional CFI. If run-time exceptions are treated as normal exceptions, each instruction that can cause a run-time exception also becomes a conditional CFI. In the CFG, a method that can throw an exception is treated as a CFI with an edge to the next basic block, as well as an edge to each catch block that may be invoked, or the exit block if a catch block cannot be determined. There can be multiple such edges, as a method may declare different distinct exception types.

If an exception is thrown, the current context taint and the exception's taint are stored. At the catch block, this taint is included in the local context taint; each catch block has a bit in the context bitmap and thus an entry in the context taint for this purpose. If the catch block is not in the same method as the throw instruction, the call stack will be unwound. Each method invocation on the stack is treated as a conditional CFI, which requires tainting variables that are written to in the current stack frame, as the instructions following the method invocation are analogous to control path that is not executed. As the stack will be unwound anyway, tainting local variables and stack elements is not required. The entire unwinding of the stack can be skipped if neither the exception nor the context were tainted.

Information can be leaked if a method that declares an exception does not throw such an exception, much like an if-statement can leak information by not executing a certain control path. Consider Listing 5:

```
boolean b = true;
try {
   leak (secret);
   b = false;
}
catch (Exception e) {}

void leak (boolean secret) throws Exception
{
```

```
    if (secret) throw new Exception ();
}
```

Listing 5: Information leak through Exceptions

In this case, the fact that the exception is not thrown conveys the fact that secret is false. Therefore, the assignment to b must also be tainted. This is handled by maintaining in *leak* a taint of exceptions that are not thrown. When the method executes, each conditional CFI that skips executing of a throw statement causes the context taint to be included in this taint. As the invocation of leak is considered to be a conditional CFI, it has a bit in the context bitmap and an entry in the context taint. This entry is set to leak's non-thrown taint, which will ensure the assignment to b is tainted. Note that after the catch block, the control paths merge, so the context taint is declassified.

'Finally' blocks, which are executed when leaving a try/catch block regardless of whether an exception is thrown, are implemented in Java as catch blocks for any type of exception. The case when no exception is thrown is handled by an explicit jump into the finally block. Therefore our approach is able to handle these blocks automatically.

3.6 Methods and Fields

When a method is invoked, the context taint is passed to that method along with any taints on the parameter values. This ensures that taints will not be lowered simply by invoking another method. When the method returns, the return value's taint is propagated back and the context taint is restored, as the calling method does not depend on any conditional CFIs in the method that stopped executing.

Taint values are stored separately for different fields in an object, avoiding label creep. Each object also has an collective object taint, which automatically includes any taint value written to any of its members. This allows a policy to quickly check if any part of an object is tainted. The policy can also set or reset this taint value.

4 Performance

In this section we briefly look at the performance overhead introduced by the taint propagation mechanism. This overhead 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. 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. Version 1.1.7 of the Kaffe JVM was used for the comparison [1] .

Table 2 summarises the result of the 'AllObjectConstruct (large assign)' microbenchmark from the jMocha benchmark suite [9] 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

[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
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
FileWriteBW in MB/s for various block sizes

their default values. As this benchmark measures only the initialisation time, which forms a very small part of the full runtime, the observed overhead is justifiable.

	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

Table 3 compares the bandwidth (in MB/sec) of writing to a file of 16M size using various block sizes for both Kaffe and Trishul JVMs. Note that here, as with the AllObjectConstruct benchmark, no taints were introduced into the Trishul system. 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 value at each CFI instance. These expensive operations introduce more overhead. A similar result was observed for the FileReadBW benchmark test too.

5 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 static analysis performed just before execution time can be cached across application runs by storing the calculated information (like the context bitmaps) in a secure, integrity protected, manner. In a similar manner, the CFGs of trusted core Java libraries used in Kaffe and Trishul, like GNU Classpath [10], could be calculated in advance and stored securely and re-used each time they are needed. A more efficient static analysis could reduce the overhead further. For example, an analysis of the branch blocks could reveal that the objects used for 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.

We are also in the process of implementing a JIT version of the system, which should offer more realistic performance.

Policy Engine - The Trishul architecture proposed here does not use a specific built-in security policy model. Rather the modular nature of the policy engine architecture allows the system to use policy engines provided by various third parties. In doing so, the architecture can not only support various policy expression languages but also policy semantics.

Native Methods - 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. There is no way for the IFC system to track the information flow within these methods. In order to avoid this, Trishul assumes that only trusted native method libraries are allowed to be accessed by Java applications. As a part of the post-build process, the hash of every trusted native library is stored within the JVM. 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 allowed to be invoked.

6 Related Work

While some of the solutions proposed [11,12] to the problem of information flow tracing do not work in practice because they rely on using CFI instruction in a very restricted impractical manner, others [13] works only if the security classes have an explicit notion of high and low, which is not always the case.

Denning [14] proposed a compile-time approach to solving the implicit information flow problem whereby the compiler adds extra instructions so that irrespective of whether the CFI branch is followed or not, the class of object acted upon within the branch is updated to reflect the information flow. Other than compile-time associated restrictions discussed earlier on, the proposed architecture was purely theoretical in nature and also depended on the use of specialised 'tagging' supported hardware for supporting dynamic binding.

Fenton's Data Mark Machine [3] was 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 RIFLE architecture [15] is a more recent system that implemented run-time information flow security with the aim of providing policy decision choice to the end user. They uses a combination of program binary translation and a hardware architecture modified specifically to aid information flow tracking, the use of which prevents it from being used on a normal machine. Beres and Dalton [16] used a dynamic instruction stream modification framework to dynamically rewrite machine code in order to support dynamic label binding. However the system not only ignored implicit information flows but also relied on enhanced hardware to perform the run-time tracing.

Chandra [17] pursues a hybrid approach similar to that used by Trishul but by instrumenting the bytecode with taint propagation code. However our exception handling is more thorough and more effective at handling control flow attacks.

Their work also does not consider the risks posed by native function in any great detail. Trishul treats labels as bitmaps, whereas Chandra treats them as integers. Our approach allows more flexible representation and manipulation of the labels by the policy engine. By using an external instrumentation process their system also relies of a bigger trusted computing bases for the accurate working of their system. Haldar et al. use similar instrumentation approach in order to enforce security policies [18]. However, the approach uses a much less accurate analysis of implicit flows and exceptions as well as support only a very coarse granularity for tainted objects.

Inlined reference monitors (IRMs) [19] 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 [20]. Because of this inability 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.'

Newsome and Song [21], among others, use the concept of tainting to track untrusted data from potentially unsafe input channels, like networks. If these data are used to perform dangerous operations, a flag is raised and execution is halted. But to extend the work to perform general policy enforcement would require considering implicit flows, which is currently omitted. JFlow [22] is one of the modern compile-time system, that uses type-based static analysis to track information flow in Java. At compile time, a special compiler uses the programmer specified labels associated with objects 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.

7 Conclusions and Future Work

In this paper we described the design and implementation of Trishul, a JVM based information flow tracing system. Trishul tackles the information flow tracing problem by using a hybrid approach by performing static and dynamic analysis of the Java bytecode. Trishul is able to perform taint tracing at the granularity of Java member fields, thus providing a very flexible architecture for data-oriented policy enforcement framework.

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. The information flow tracing mechanism described in this paper forms one of the core component of Trishul system architecture. We are in the process of implementing the pluggable policy engine module section as well as a language for the policy engine writer to develop these policy engines.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

[13] 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.

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

[15] 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.

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

[17] D. Chandra, *Information Flow Analysis and Enforcement in Java Bytecode*, Ph.D. Thesis, University of California, Irvine, 2006.

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

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

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

[21] J. Newsome and D.X. Song, *Dynamic Taint Analysis for Automatic Detection, Analysis and Singatue Generation of Exploits on Commodity Software*, 12th Network and Distributed System Security Symposium, 2005.

[22] 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.