Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine

Deepak Chandra and Michael Franz
University of California, Irvine

Abstract

We have implemented an information flow framework for the Java Virtual Machine that combines static and dynamic techniques to capture not only explicit flows, but also implicit ones resulting from control flow. Unlike other approaches that freeze policies at time of compilation, our system truly separates policy and enforcement mechanism and thereby permits policy changes even while a program is running. Ahead of execution, we run a static analysis that annotates an executable with information-flow information. During execution, we then use the annotations to safely update the labels of variables that lie in alternative paths of execution while enforcing the policy currently in place. Our framework doesn't require access to source code and is fully backward-compatible with existing Java class files. Preliminary benchmark results suggest that the run-time overhead of information flow techniques such as ours is well within acceptable range for many application domains.

1. Introduction

The traditional definition of a Trojan horse is a program that exploits read access to a secret and simultaneous write access to a public channel to leak the secret "downwards". In that sense, your network-connected home PC may have the ultimate Trojan horse installed on it already—the web browser. You have to **trust** your browser not to leak your banking secrets to the kids' web space provider when you check up on their online activities, but you have no way of knowing whether that trust is warranted. The existence of executable content (such as Java and JavaScript) within web pages only exacerbates the problem, because the functionality of the browser can be extended dynamically. Unfortunately, there already have been incidents in which one web site has exploited an error in the browser to steal secrets destined for an unrelated second web site viewed in the same browser.

At the server end of things, the situation is possibly even more serious. As more and more services (banking, e-government, etc.) are migrating to the Internet, the providers of these services are constructing web front-ends that may contain exploitable errors. A common mistake is relying on a script on the client's web browser to validate user input, when an attacker can actually send maliciously crafted input to the server directly (using the `http` protocol).

The original emphasis in the Java infrastructure has been on protecting the host system from malicious actions of downloaded Java programs. This protection is provided by way of type-safety and memory safety and by maintaining the same visibility and data access abstractions in the virtual machine as they existed in the Java program source text. Much less emphasis has so far been placed on the integrity and confidentiality of the *information* handled within the virtual machine. In particular, Java has discretionary access controls rather than mandatory ones—once that access to information has been granted, the holder of that information can propagate it completely unchecked. The aim of our research is to improve the safety of Java-based server front-ends and client-side browser extensions, by providing fine-grained and flexible information-flow controls while staying fully backward-compatible with the Java bytecode format. We retrofit information-flow controls on existing Java bytecode programs without requiring recompilation or indeed any access to source code.

Our system provides completely dynamic policies that can be changed even while a program is already running. This is in contrast to previous approaches such as Myers' Jflow and Jif [30, 31] that extend the Java language with statically checkable information-flow annotations. At compilation time, the information-flow policy then becomes "frozen". In Erlingsson and Schneider's approach [16, 17], a security automaton [33] is in-lined into each program prior to execution, which again "freezes" the security policy; furthermore, this solution is limited by what is decidable by a security automaton. In earlier work by Haldar et al. [22, 21], a label is added to every object within the Java Virtual Machine (JVM). This label is then used for purposes such as taint propagation, but the system does not consider information flows that are carried by control flow and hence can-

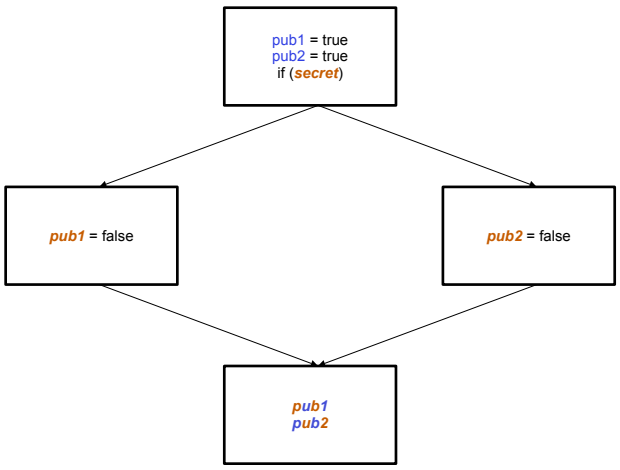

Figure 1. Implicit information flow from *secret* to *pub1* and *pub2*.

not be used to implement a full-fledged multi-level security (MLS) scheme [15].

The work presented here provides information labeling at a finer granularity than Haldar et al. and additionally also correctly handles information flow through control flow. It is thereby in the good company of pioneering previous work on implementing MLS using virtual machines [32, 39, 25], but rather than doing so at the level of a hardware-close virtual machine monitor, our implementation is situated at the richly typed level of the JVM. Our implementation is noteworthy in that it combines static (ahead of execution) analysis techniques and dynamic (concurrent with execution) enforcement of security policies, and in not requiring any access to source code.

The remainder of this paper is organized as follows. First, we briefly present the basics of information flow and explain what problem we are solving (Section 2). Then, we present our *hybrid* technique that combines static analysis with dynamic enforcement (Section 3). This makes our technique less conservative than current information flow analysis techniques while it is simultaneously safe, fast, and flexible. Section 4 presents benchmark results for our implemented prototype. Section 5 presents related work. Finally, we conclude in Section 6.

2. Problem Statement

We say that information flows from A to B when B can observe changes to its environment that depend on the value of A. The most straightforward way in which information can flow in a program is through an assignment. This is also called an explicit flow. For example in the statement $a = b$, there is flow of information from b to a as one can know the value of b by looking at the value of a. Another explicit flow, albeit with some loss of information, would be in $c = a + b$. Here information flows from both a and b to c and some information is lost due to the arithmetic operation. All assignment statements $lhs = expression$, including parameter and return value passing, lead to a flow of information from all operands in $expression$ to lhs.

Information can also be propagated through the control flow of a program. For example in Figure 1, there is no direct assignment from *secret* to either of the two other variables, but still at the end of the execution of the code fragment, one can infer what the value of *secret* was. Information can flow implicitly in two ways. By following the branch, we can infer what the branch-controlling variable was. In Figure 1, after observing that the value of *pub1* has changed, one can infer that *secret* is true. The traditional solution to this problem is by attaching a security label to the program counter as well, and coercing the labels of all assignment targets to the least upper bound of their current label and that of the program counter. Hence, in our example, *pub1* would become a secret as well because it is being modified in a program region guarded by a secret control variable.

More subtly, observing the **non-execution of a branch** can leak information about the branch condition as well. For example, by observing that the value of *pub2* has not changed when control flow re-joins, one can infer that *secret* is false. Hence, in this example, observing just one of the variables *pub1* or *pub2* would be sufficient to infer the state of *secret*, even though each of the two control flow alternatives modifies only one of the variables.

This is exactly the problem that makes it difficult to control information flow by strictly dynamic means, i.e., by following only the path through the program as it is actually taken. In order to inhibit flows that result from assignments in alternative branches, one needs to consider all alternative branches simultaneously. We solve this problem by using a combination of static analysis and dynamic techniques to handle control-dependent information flows. We perform static analyses ahead of actual execution. These analyses result in annotations that enable the virtual machine to later insert compensating tag instructions into alternative paths whenever a variable is modified along just one path. The analysis also determines and annotates the earliest point that the program counter's label can be lowered again after a control flow join.

Consider the example from Figure 1 again as we have presented it in Figure 2. Here, branching on *secret* raises the label of the program counter to the least upper bound of its current value and that of *secret*. Every variable that is modified has its own label coerced to the least upper bound of its current label and the label of the program counter. For

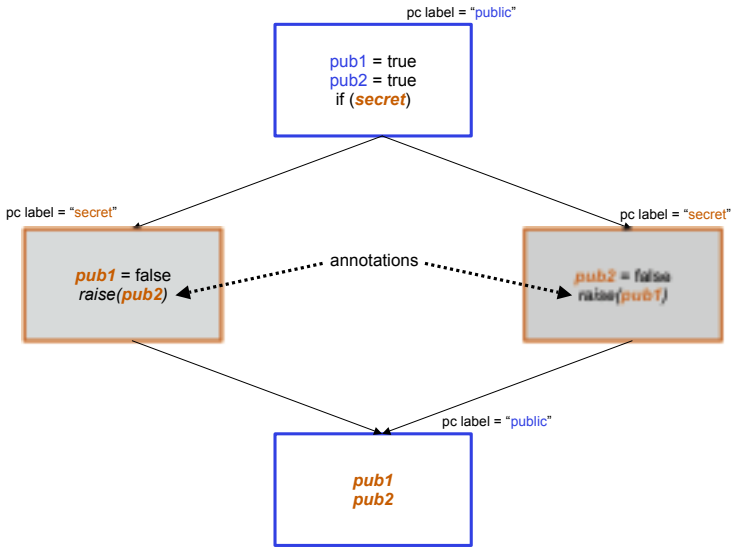

Figure 2. Hybrid analysis—ahead of execution, information is gathered that later permits to update labels as if all branching alternatives had been considered in parallel. Additionally, the analysis identifies and annotates the earliest point at which the program counter can be lowered again.

example, the assignment to the non-secret variable *pub1* in the left side of the branch will raise the label of *pub1* to that of *secret* because the program counter is secret, preventing a leak of the contents of variable *secret* via *pub1*. In spite of the coercion of *pub1*'s label, we would still be able to infer the value of *secret* after the control flow re-joins at the end of the if statement by observing the non-secret variable *pub2*. This is prevented in our implementation by automatically inserting compensation code, so that when any variable is modified in *any* branch, its label is coerced in *all* branches. When the control flow re-joins, the program counter's label can be restored to the value it had before the branch. Note that *pub1* and *pub2* remain classified past this control-flow join. They will be declassified only when the next public value is written into them, overwriting whatever secret may still be contained in them.

Unfortunately, the situation in real-life Java programs is hardly ever as simple as in our example in Figure 2. In Figure 2, we know exactly which variable in the *alternative* control path is affected, and hence we can insert a compensating operation that raises the label of that variable. However, in typical Java programs we often don't know exactly what variables might be affected in alternative control paths, but have to deal with a congruence class of potentially affected variables. Any member of this congruence class could be affected—so we have to coerce the labels of **all** of them in order to regulate information flows.

For a better illustration of this situation, consider Figure 3. While following the actual flow of control, we know exactly which object is modified and hence needs to be coerced to the same label as *secret*. However, without actually executing the alternative branch, we cannot be so sure about what object might have been affected in that other program part. In many cases, we also cannot just execute the alternative branch to find out, because that might introduce subtle side effects. There might also be a combinatorial explosion if we attempted to execute all possible paths always. So if we want to safely exclude illegal information flows, we have to make a conservative assumption and raise the label of all variables that could possibly be affected. In this case, that means raising the label of the g field in all objects that are of class *foo*.

In the remainder of this paper, we report on our implementation that does just that. As with almost all static analysis techniques, a compromise has to be made between the precision of the analysis and its cost. For example, a less precise analysis in our example of Figure 3 might have changed the label on all fields of all *foo* objects instead of discriminating between the f and g fields. A more precise analysis might analyze all variable bindings outside of the method and determine that p and q actually point to the same object always. If $p = q$ always holds inside of m, then in each alternative path, the congruence class of possibly affected variables contains only a single element. Effectively, this re-creates the situation of Figure 2.

Our analysis does not cover covert channels [26] in any way, nor does it try to inhibit them. For example, one might perform zero or one million iterations of a loop depending

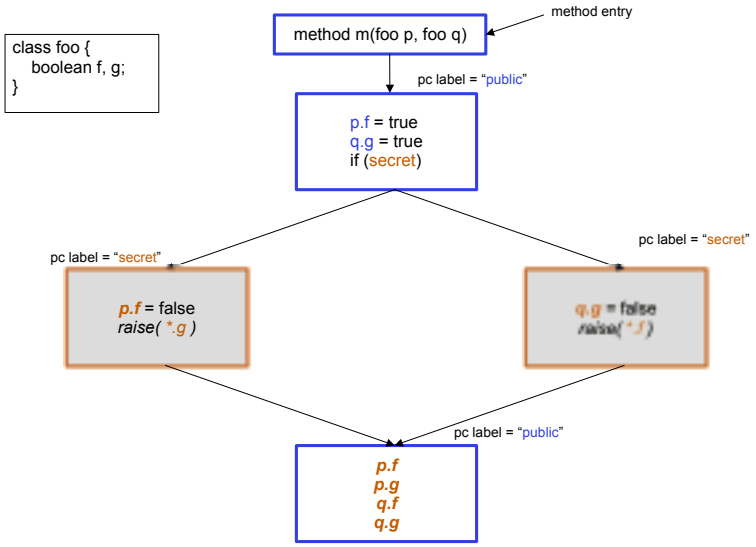

Figure 3. In this example, we no longer know exactly which variable is affected in the alternative branch, so we have to be conservative and coerce a whole congruence class of storage locations. Note that p and q may even point to the same object here.

on the value of a secret variable, and then infer the secret value based on how long the computation took. Such considerations are explicitly outside the scope of our project and will have to be dealt with using techniques such as randomized clock speeds, etc.

3. Hybrid Analysis and Enforcement

Our hybrid information flow technique finds a synergy between static and dynamic approaches. It is primarily a dynamic technique that tracks the flow of information at runtime, but it uses static analysis annotations to make safe decisions about implicit information flows. In particular, these annotations enable us to consider flows in alternative paths of control without having to actually execute those paths. Our analysis is safe while retaining the flexibility of a purely dynamic approach, namely the ability to change the policy while the program is executing.

Another important feature of our approach is that we did not change the Java class file format in any manner. We remain fully backward-compatible with the vast amounts of Java bytecode that already exists. Our system does not require program source code or annotations from the programmer. This makes our system useful in many scenarios where the source code many not be accessible.

We have implemented our analysis using bytecode instrumentation with very little changes to the underlying Java virtual machine. This makes our implementation easily portable across many JVMs. In the remainder of this section, we will give an overview of our implementation.

3.1. Terminology

- **security label** is a runtime entity that indicates the security level of a value. Each field and local has a label associated with it that indicates the label of the value stored. For conciseness, a label for a local or field named x is represented as $\underline{x}$.
- $\underline{\boldsymbol{a}} \bigcup \underline{\boldsymbol{b}}$ represents the label of a value derived from a and b. It is at least as restrictive as $\underline{a}$ and $\underline{b}$. In a lattice of labels it is the least upper bound (LUB) of the two labels.
- $\underline{\boldsymbol{pc}}$ is the label of the information implicit in the control flow at any point. It is also called the "program counter label".
- **declassification of $\underline{\boldsymbol{pc}}$** is a process of reducing the value of $\underline{pc}$. This may happen when there is reduction of information in the control flow, which happens at control-flow joins.

3.2. Security Labels

Security labels track the sensitivity of the information in an entity. Our analysis tracks information at the level

of fields (instance variables) and local variables, which is a very fine level of granularity.

A *label hierarchy* defines the labels for a policy and the relationship between them. Our analysis supports label hierarchies that are finite total ordered set with respect to an ordering relationship $\leq$. Such a fully ordered label hierarchy is similar to the DoD clearance level structure. We use a fully ordered label hierarchy instead of a more generic lattice structure for reasons of efficiency. Since our analysis dynamically calculates labels for each value computed, efficient implementation of all label operations is critical for the overall performance of the analysis. Many of our implementation decisions are influenced heavily by efficiency issues. Throughout the paper, we will be discussing the implementation decisions we made for performance reasons.

We implement labels using integers in which the position of the most significant 1 defines the value of the label. This scheme helps us implement the operations $\leq$ and $\bigcup$ very efficiently. Since Java integers are 32 bits, we can currently have only 32 labels. However, this can be easily be extended to 64 bits if we use long-integers instead.

Our analysis is dependent on the precision of the underlying static analysis. This was mentioned in our discussion of Figure 3. Our current implementation of the static analysis represents objects by their allocation sites. This means that one static object can potentially represent many runtime objects simultaneously (for example, consider a `new` statement inside a loop). The resulting loss of precision stems from the points-to analysis that we use. Similar limitations are typical for most points-to analysis algorithms. This is not a limitation of our information-flow analysis, but of the underlying points-to analysis. Any improvements to the points-to analysis will directly benefit our information-flow analysis.

In order for many runtime objects to share a common label, we need to maintain the labels of fields separate from the corresponding objects, in a global array. There are actually two global arrays, one for static fields and one for instance fields. Since a static field is shared between all instances of a class, we need only one label per static field. Static fields are resolved at compile time and hence accessing the label of a static field is a simple lookup into the static label array at runtime.

The instance field array stores the labels of instance fields of allocation objects. Every object has an object offset number, which is the index in the instance field array from where the labels of its fields are stored. Each field in a class also has a field offset number to indicate the offset from the beginning of the object where its label is stored. Hence, the label of an instance field whose object offset number is 10 and field number is 2 is stored at the (10+2) location in the array. To reduce the overhead of instance field label look up, we statically determine the field offset. Due to Java's polymorphism, this is complicated since a reference can point to an object of a derived class. We overcome this problem by laying out the fields in the array in ways similar to how compilers handle dynamic dispatch of fields in polymorphic languages.

Even though the field offset within the object can be determined statically, the actual object that the reference is pointing to is not known until runtime. Hence to access the label of a field, say *foo.pub1*, we first need to know the offset of *foo* in the instance field array. To do this, we instrument all classes to have an additional field that stores this offset. We call this field *_mac_obj_off*. The value of the offset field is determined at runtime by the allocation site instantiating the object. We instrument all allocation sites so that they appropriately pass the allocation number to the appropriate constructors. We also instrument all the constructors in a class to initialize the *_mac_obj_off* to a value communicated by the allocation site. We use a global class, which we describe later, to communicate the allocation number between the allocation site and the constructor. Hence the runtime address calculation for the label of *foo.pub1*, is *foo._mac_obj_off* + *field_no*. Therefore to access the label of an instance field requires a field access and an array access.

To make this calculation more amenable to optimization, we declare *_mac_obj_off* as `final`. A final field, once initialized, is guaranteed to have the same value always. Hence the just-in-time compiler can treat the calculation for the label as a constant. Our experiments have shown that this has a significant effect on the run-time performance of instance field accesses.

3.3. Security Policy Specification

The security policy is responsible for specifying sources, sinks, and declassifiers for labels. "Sources" are the sources of sensitive information in the program. The policy specifies them as entities whose labels will always be as strict as some label constant. For example, a policy may specify that the label of the return value of the method `getPassword()` is always at least as strict as *secret*. Typically, sources are return values of methods but they can also be fields in a class. Sources seed the system with labels of sensitive information, which are then propagated to other parts of the system. A policy also defines the default label used by the system at initialization.

Sinks are methods whose side effects may be observable externally to the program, such as a file read/write. A policy specifies all such methods and also the most restrictive label that is allowed to be passed into such a method as a parameter. Any call to a sink with a parameter more restrictive than the specified label is blocked by runtime mechanisms.

Bear in mind that our system retrofits information-flow

constraints **on existing bytecode** *after the fact*. In practice, this is done by explicitly specifying labels only on relatively few data sources and sinks, most of which will be associated with library classes. Our system will then automatically prevent information flows that violate the current policies. For example, it can prevent passwords obtained as a result of calling `getPassword()` to be written to the file system—without any modification or recompilation of the program on which this policy is being enforced.

Policies are also responsible for specifying declassifiers. Declassifiers are places in the program at which a policy dictates to lower the label of a variable. This may lead to a leak of information, but in some circumstances the leak of information is desirable. For example, in a password checking program, the return value of the program depends on the actual password, and hence transmitting the information is a leak of information. However, returning this small amount of information whether or not the password matches is actually the purpose of the program, and hence a policy should specifically allow it. The policy specifies such declassification points. Such declassifiers are different from automatic declassification points, in which the analysis determines automatically that is safe to lower the program counter label.

Our method of policy specification is different from program annotations in systems such as Myers' Jflow and Jif [30, 31] since it is centralized and cleanly separated from the code. Also, a well specified policy can be independent of a program. For example, in the Java core libraries there are only very few methods that directly access the file system. If a policy can specify this narrow funnel, then any program accessing files can use this policy. Currently we specify policy by writing code in Java itself. This gives us a lot of flexibility to experiment with many policies. However, in the future we want to define a declarative policy specification language which the instrumentation code understands, so that it can automatically instrument the classes.

3.4. Dynamic Analysis

Dynamic analysis uses runtime labels to track explicit flows of information throughout the execution of a program. It also tracks implicit flows by raising the value of $\underline{pc}$ on every conditional jump to as high as the label of the condition. It however needs static analysis annotations to lower the value of $\underline{pc}$. Without static analysis, our approach would be too conservative as once the value of $\underline{pc}$ were raised, it would never be lowered. The aim of the dynamic analysis is to create runtime entities for storing labels for each local variable and field and tracking the flow of information through the execution of a program.

The dynamic analysis instrumentation is intra-method, i.e. it does not need information about other methods. The instrumentation is divided into two phases. The first phase instruments all reachable methods from the main method with instructions to track the flow of information. The second phase instruments the sources, sinks, and declassifiers appropriately to initialize tags, stop illegal flows, and to lower the values of labels.

During the instrumentation phase, we do not try to optimize redundant label computations. We leave optimizations for a later phase, where we rely on common compiler optimization techniques. We tailor our instrumentation so that the compiler can easily discover optimizations opportunities. In the rest of this section we will describe how we instrument various types of statements to track information flow using labels.

3.4.1 Method Calls

Method calls present a challenge as we have to communicate labels between the caller and callee. For every method call, the following three communications are required between the caller and callee: (1) pass the value of $\underline{pc}$ to the callee method, (2) pass the labels of all the parameters to the callee method, and (3) get the label of the return value from the callee method.

For the purpose of sharing information between the caller and callee methods, we define a public class. We call this class as the "global class". The class consists of a number of public fields with each field being used to store the label of a formal parameter in a method call. The number of fields in the class is determined during the analysis, i.e. the number of fields in the global class is equal to the maximum number of parameters in any method that we analyse. This ensures that our global class remains as small as possible. Since method calls are frequent, it is desirable that the class remains in the processor's data cache. The $\underline{pc}$ is passed in a similar fashion using the field pcTag.

We instrument every call site during the analysis to assign labels of actual parameters to appropriate fields in the global class. If a method returns a value, then the label of the return value is assigned the value in the pcTag field of the global class. It is the responsibility of the callee method to have assigned the correct label in the pcTag field before returning to the caller.

3.4.2 Method Entry

At a method entry, the analysis has the following tasks to perform: (1) create labels for all the locals in the method, and (2) initialize all labels for the parameters. We create a label for each local in the method by adding the same number of integer locals to the method, with each new integer local acting as a label. Once the labels have been added, we instrument the method entry to initialize the labels of the parameters using the global class. This procedure is just the opposite of when the method is called. We initialize the

labels with the respective fields in the global class. We also initialize the $\underline{pc}$ appropriately.

We instrument constructors slightly differently from other methods. A constructor is also responsible for initializing the *_mac_nodeNo* field of a class. This field has the index in the instance field array where the labels of the instance fields of the object are stored. The value of *_mac_nodeNo* depends on the allocation site that the constructor is called from. The allocation site at runtime communicates this by assigning the value to the *nodeNo* field in the global class. The constructor then appropriately uses this to initialize the *_mac_nodeNo* in the object.

3.4.3 Array Label

We label arrays at the granularity of an entire array i.e. we have one label for the entire array. The value of the label is at least as restrictive as the most restrictive label of any element in the array. This means if an array has an element with label *secret*, then the label for the array will also be *secret*, and all the elements in the array will be treated as having label *secret*. The coarser labeling leads to a loss of resolution, making the analysis less precise than if we had tracked the label of each element in the array separately. However, it is not statically possible, in a general case, to determine *which element* in the array was accessed for an array access. Just consider the statement `a = foo[random(seed)% foo.length]`. In order to be conservative, we need to assume that an array access can refer to any element in the array, and hence have to work with the strictest label of all the elements assigned to the array.

3.4.4 Assignments

An assignment statement always represents a flow of information from a right-hand side expression (RHS) to a left hand side (LHS). Afterwards, the LHS identifier should get a label which is at least as restrictive as the label of RHS expression. If, after the assignment, the LHS identifier does not contain any information about its previous value, then the least restrictive label of the LHS variable which is as restrictive as the label of RHS is the label of the RHS. For all assignments other than an assignment to an array element, the label of the LHS is the label of the RHS expression. The label of an array represents all the elements in the array, hence when there is an assignment to an array element (of unknown index), the label of an array should be as at least as restrictive as its current label and the label of the RHS expression.

3.4.5 Expressions

An expression consists of one or more operands and an operator acting on it. For unary expressions, the label of the operand is the label of the expression. For expressions with multiple operands, the label of the expression is the $\bigcup$ of the labels of the operands.

3.4.6 IF Statements

If statements are responsible for an implicit flow of information. We therefore raise the value of $\underline{pc}$ to be at least as restrictive as the label of the branching condition i.e. $\underline{pc} = \underline{pc} \bigcup \underline{\text{condition}}$. A condition is a type of an expression, and its label is calculated just like any other binary expression.

3.4.7 Unconditional Goto and Break Statements

An unconditional goto statement or a break statement do not lead to an explicit flow of information. They purely affect the control flow of a program and may lead to some implicit information flow. Static analysis takes such implicit flows into account.

3.4.8 Exceptions

For dynamic analysis, we consider exceptions as unconditional jumps, either to a catch statement in the method or to method exit if the method does not handle that exception. There is no implicit or explicit flow of information. Exceptions purely affect the control flow graph of the method, which affects the static analysis. We discuss the effects of exceptions to the analysis and how they can be used as covert channels in some more detail below.

3.4.9 Return Statements

Similar to exceptions and unconditional goto statements, return statements are also considered as unconditional jumps to the exit block of a control flow graph. It is the responsibility of the method to assign the label of the return value to pcLabel of the global class before returning; therefore statements are instrumented appropriately to do this at all method exits.

3.5. Static Analysis

Purely dynamic information flow techniques do not have information about all paths of execution to make safe decisions about declassifying $\underline{pc}$. This makes them too conservative. Hybrid information flow analysis uses a dynamic technique without being overly conservative by taking help from static analysis. Statically collected information about all branches of execution is communicated to the runtime mechanism. The runtime mechanism then declassifies $\underline{pc}$ whenever it is safe to do so. Before declassifying the $\underline{pc}$, it raises the labels of all the variables with implicit information to be as high as the $\underline{pc}$. This ensures that all

entities that have implicit information about the condition have adequately high labels before the value of $\underline{pc}$ is lowered. This approach can also be thought of as emulating the effects of calculating all labels on all branches. Hence an observer will not be able to determine which branch got executed.

Static analysis provides the following information to the runtime mechanisms: (1) all the variables that have implicit information about a condition, and (2) all points in the control flow graph where there is no implicit information about a condition. These points are called declassification points for $\underline{pc}$.

We use the immediate postdominator of a conditional statement as a declassification point. To understand why an immediate postdominator is a safe declassification point, consider how the control flow of a program can be used for implicit flows. If the control flow of a program is a straight line with no conditional jumps, then all the nodes will be executed and the knowledge of an execution of a node will not indicate any additional information about the program. However on conditional jumps, the value of the condition determines which nodes get executed and therefore knowledge of whether a node executed can be used to infer the value of the condition. There is an implicit flow of information from the condition to all the nodes whose execution is affected by the condition.

By definition, a postdominator node always gets executed irrespective of the value of the condition, hence there is no implicit information about the condition in the control flow at that point. Therefore $\underline{pc}$, which tracks the label of implicit information in the control flow graph can be recalculated at that point without the label from the condition. We instrument code at the immediate postdominator node to calculate the new value of the $\underline{pc}$. Recalculating the $\underline{pc}$ may not always lower its value, since the control flow may still contain information about conditional jumps that occurred even earlier in the control flow graph.

We use a simple backward dataflow analysis to find postdominator information about the graph. The dataflow equations for the analysis are given below. This algorithm is derived from the dominator algorithm described in [12].

$$Postdom(n_0) = n_0$$

$$Postdom(n) = \left(\bigcap_{p \in successor(n)} Dom(p) \right) \cup n$$

Static analysis is also responsible for identifying and instrumenting code to ensure that the labels of all the variables that have implicit information about the condition are raised as high as the label of the condition before declassifying $\underline{pc}$. Implicit information flows from the condition variable to all the variables that are assigned in all the branches of the condition.

We use side-effect analysis to identify all the variables that can be assigned under the condition. Side-effect analysis is a whole program analysis that computes all possible side-effects for each statement. This analysis in turn is dependent on points-to analysis which finds all possible objects that a particular reference may point to at run time. The precision of the side-effect analysis has a big effect on the precision of our static analysis. However, we implemented our analysis in a way that a new side-effect analysis can be easily plugged in, with very little changes to the code.

After we identify all the variables that can change under a condition, we instrument the code to increase their labels to be at least as strict as program counter label in the immediate postdominator node. Once we have raised the labels of the affected variables, we declassify the $\underline{pc}$. Declassifying before this point would be unsafe.

3.6. Unchecked Exceptions

A limitation of our analysis is its inability to handle implicit flows through unchecked exceptions. Unchecked exceptions are exceptions that are not caught within the method raising the exception. In Java, many bytecode instructions can potentially throw an exception such as null pointer, divide by zero, and array out of bounds.

From a static analysis point of view, an unchecked exception is an additional edge in the control flow graph from the instruction throwing it to the method exit. If not handled properly, it can leak an arbitrary amount of information through implicit flows. In the following Java example, one can know the value of *secret* by counting the number of *'s that are printed.

```
i=0;
while(true){
   i++;
   if (secret == i)
       throw MyException;
   print('*');
}
```

Unfortunately, handling every unchecked exception would make our analysis overly conservative as there will be an edge from every bytecode that can throw an exception to the exit block. Since there are so many bytecodes that can potentially throw unchecked exceptions, most of the nodes in the control flow graph would have exit block as their immediate postdominator. This limitation is an inherent drawback of using Java. It applies equally to information-flow systems extend the Java language such as Myers' Jflow and Jif [30, 31].

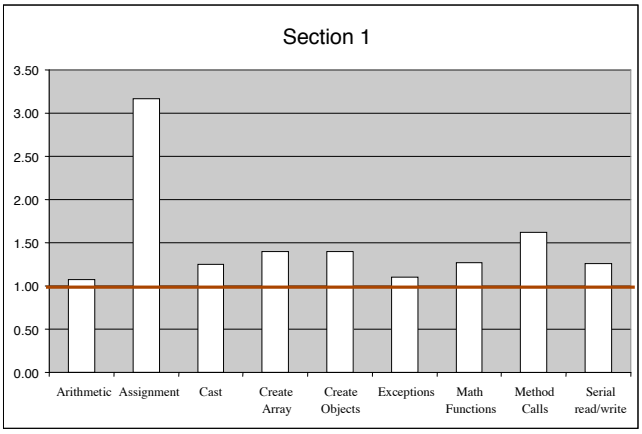

Figure 4. JavaGrande Section 1, normalized slowdowns relative to the unmodified benchmarks.

4. Implementation and Results

We have implemented the described hybrid information flow system for Java using the Soot framework [36] for the underlying bytecode analysis. We used IBM's JikesRVM as our virtual machine platform. To assess the performance overhead of our analysis, we benchmarked the performance of the resulting prototype using the JavaGrande [11] Version 2.0 benchmarks. These benchmarks consists of three sections. They are:

- Section 1 – Low Level Operations: low level operations such arithmetic, assignments, casts, method calls, object creation, math functions, serial read/writes, and exception handling.
- Section 2 – Kernels: operations such as FFT, Fourier coefficient analysis, and matrix multiplication, which are frequently used in scientific applications.
- Section 3 – Large Scale Applications: large-scale applications that use large amounts of processing, I/O, network bandwidth or memory. The applications included are computational fluid dynamics, monte carlo simulation, and 3D ray tracer.

Calculating labels at runtime for a fine-grained analysis can incur a substantial overhead. We were conscious about this and structured our code in ways that made optimizations opportunities explicit to the compiler or the JIT. For example, we laid out objects in such a way that the field offset can be resolved at compile time. Also, for instance fields, the object offset is made `final` so that the compiler can optimize away the calculation of the instance field offset.

During the instrumentation phase, we focused on the semantics of label calculations and did not optimize for any redundant label calculations. In a later stage, we ran standard compiler optimizations that are built into the Soot framework. Since we optimized instrumented bytecode using Soot's optimizations, we also performed the same optimizations for the unmodified benchmarks to present a fair comparison.

We ran the benchmarks on a Pentium-4 processor at a clock frequency of 1.7 GHz, running the Linux kernel 2.6.12 with 1 Gigabyte of memory. The JikesRVM image was created using the "production" configuration. Both the modified and unmodified benchmarks were optimized using the soot -O flag, which provides simple intraprocedural bytecode optimizations such as common subexpression elimination.

Figure 4 shows the result of the Section 1 micro benchmarks. The diagram presents the slowdown as a factor of the unmodified benchmarks normalized to 1. The calculation is α/β, where α is the operations/second of unmodified benchmarks and β is the operations/second for instrumented code with information flow analysis. To interpret the results, 'x' means that the instrumented application with information flow analysis ran 'x' times slower than the unmodified application. Smaller numbers are better.

Most of the micro benchmarks have slowdowns of 25-50% relative to the original code. Assignment as expected has the highest overhead and is 3 times slower that the original. This is because assignment is where explicit flow of information happens, and hence there is a need to recalculate the label for the right hand side. For every assignment, the following two calculations are required: (1) calculation of the RHS label, and (2) OR'ing the RHS label with the $\underline{pc}$, to ensure that the resultant label is higher than the program counter label.

Calculation of the RHS label depends on whether it is a field and whether it is local. For locals, the label access is inexpensive since it is stored in another label. However,

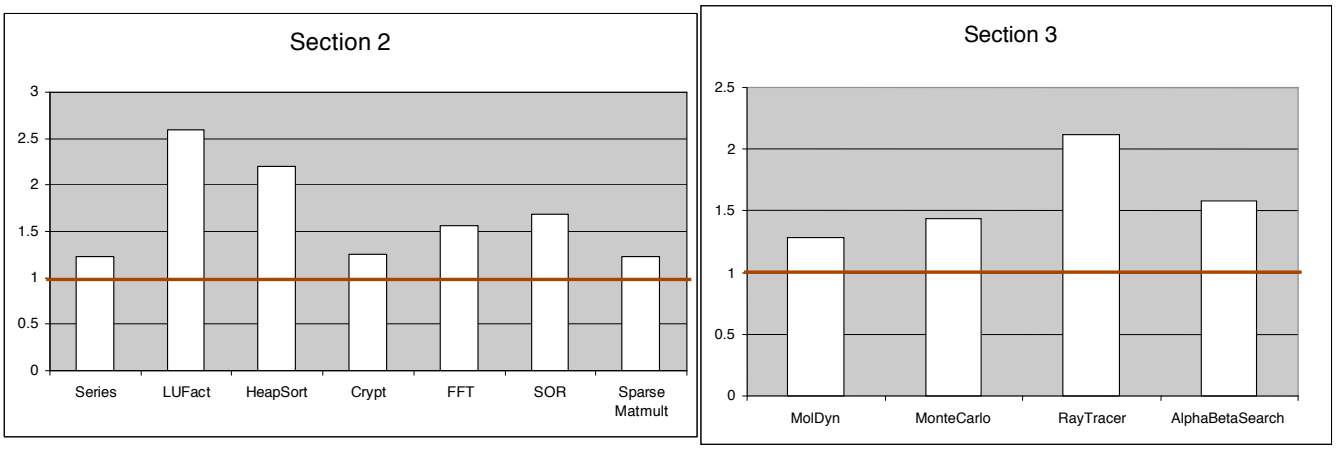

Figure 5. JavaGrande Sections 2 & 3, normalized slowdowns relative to the unmodified benchmarks.

for field accesses depending on whether the field is static or an instance field, the label accesses are different. For static fields, it is a lookup into the static label array as the location is statically determined. For instance fields, there is a field lookup to find the object offset, an addition with the field offset, an array access, and an OR operation with the reference label. Therefore field accesses are expensive, making assignments expensive in total.

The left part of Figure 5 shows the overhead for Section 2 applications. These are routines that carry out specific operations frequently needed by scientific applications. The overheads range from 23% to 159%.

Finally, the right part of Figure 5 shows overheads for large, more realistic applications. These applications are a mix of computationally intensive and I/O bound applications, and try to represent real world applications. The overheads range from 25% for the molecular dynamics to 100% for the ray tracer application.

These overheads are substantially lower than we had expected. A slowdown of a factor of two for a realistic program, while noticeable, may be perfectly tolerable in many application domains when taken in the context of added information-flow security.

5. Related Work

Bell and LaPadula [7] pioneered the use of a state machine to model security policies that specify security levels for data, and access rules for users with different clearance levels. Every event in a system is mapped to a transition in a corresponding state machine. Safety of a system is ensured by allowing transitions only to secure states. A secure state is defined as one in which the user has adequate clearance, as defined by the security policy, to access the data. The model also ensures data integrity by only allowing processes or users with the same clearance level to perform destructive writes to an object. Non-destructive writes are allowed to low-level clearance processes as long as this does not lead to an information leak.

Bell and LaPadula used an ordered set of labels such as unclassified, classified, and secret. Our work uses a labeling scheme that is similar to what Bell and LaPadula proposed, i.e. a fully ordered set. The decision to use a scheme like this over a lattice structure was for efficiency purposes, since we calculate the label at runtime.

Fenton's Data Mark Machine (DMM) [20] was an abstract machine that applied the concepts that were proposed by Bell and LaPadula. The problem with these early works in information flow was that they were completely dynamic and hence were not good at detecting implicit flows.

Denning and Denning [13, 14] were the first to point out that the information flow property should be enforced statically to contain label creep. They also proposed using a lattice structure for label hierarchy, which is more expressive than the label hierarchy of Bell-LaPadula. Since their analysis was completely static, it could afford to perform expensive label computations on a lattice without impacting the runtime performance. Lots of later work in information flow [19, 27, 28, 29, 3, 4, 6, 10, 9] was motivated by Denning and Denning's work—primarily trying to formalize the ideas they had proposed.

More recently, the non-interference property has been studied and formulated in terms of type systems, particularly in pure λ-calculi [1, 2, 23]. Volpano et al. [37, 38] formalized the soundness of Denning's analysis by developing a type system that is equivalent to the rules proposed by Denning. They then proved that this type-system observes non-interference. Banerjee and Naumann [5] extend the scope of Volpano's work to encompass data-flow via mutable object fields and control-flow in dynamically dispatched

method calls. The non-interference property is proved in much richer context with constructs of pointers and mutable state, private fields and class-based visibility, dynamic binding and inheritance, casts and type tests, and mutually recursive classes and methods. Bernardeschi and et al. [8] use type-based abstract interpretation (which is similar to bytecode verification) to prove information flow safety of Java bytecode. They, like Denning, handle implicit flows and make use of the immediate post-dominator relation to declassify the security label of the execution context. Our approach is different from their purely static analysis as we use both dynamic and static techniques, which makes the analysis more flexible and precise.

Several research projects apply static analysis to C programs. Evans' Split static analyzer [18] takes as input C source code annotated with "tainted" and "untainted" annotations. This is accompanied by rules for how objects can be converted from one into the other, and which functions expect what kinds of arguments. Shankar et al [34] use a similar approach in which C source code is annotated, but they use type qualifiers instead. The WebSSARI [24] project analyzes information flow in PHP applications statically. It inserts runtime guards in potentially insecure regions of code. It differs from approaches such as Myers' Jflow and Jif [30, 31] in that it does not require source annotations.

RIFLE [35] is a system that tracks information flow dynamically using a combination of hardware and software. The underlying hardware architecture is modified to explicitly track information flow labels on words. At load time, binaries are rewritten from the standard instruction set to a new one that also appends security labels to instructions. This translation also does a data-flow and reachability analysis on the binary. It converts implicit flows to explicit flows that can then be tracked by the architecture. This is the first approach that uses a combination of static and dynamic techniques to perform information flow analysis.

Our analysis comes closest to RIFLE since we also use combination of static and dynamic information flow analysis. However, there are major differences between RIFLE and our system. Our solution is software-only and does not require modifications to the underlying hardware architecture. RIFLE analyzes native binaries while we use Java bytecode. Due to the very low-level semantics of native binaries, their static analysis is far more conservative than ours. Java bytecode has much higher level semantics and stricter guarantees, which helps our analysis to more precisely reason about program behavior.

6. Summary and Conclusion

Our information-flow framework demonstrates that by adding statically gathered information to dynamic information flow techniques, one can make the dynamic analysis more intelligent about implicit flows while still retaining the flexibility of a dynamic analysis.

We took a lot of care during the implementation of our analysis to make sure that the runtime overhead is minimal. The results show that even for large applications, execution times no more than double. Such an overhead may be perfectly acceptable in many contexts that are particularly security sensitive.

While we tried to minimize the impact of label calculations, we still have relied on standard compiler optimizations. In the future, we want to investigate compiler techniques specifically targeted at reducing the overhead of label computations. We are confident that the cost of information flow for the Java Virtual Machine can be reduced even further using dedicated compiler optimizations. We hope that eventually, the Java community will embrace information flow techniques with the goal of making Java even safer than it is today.

Acknowledgement

This research effort was partially funded by the United States Homeland Security Advanced Research Projects Agency (HSARPA) and Air Force Research Laboratory (AFRL) under agreement number FA8750-05-2-0216, and by the National Science Foundation (NSF) under grant CT-ISG-0627747. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of HSARPA, AFRL, NSF, or any other agency of the United States Government.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, 1999.

[2] M. Abadi, B. Lampson, and J.-J. Levy. Analysis and caching of dependencies. In *ICFP '96: Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 83–91, 1996.

[3] J.-P. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *CSFW '93: Proceedings of the 6th IEEE Computer Security Foundations Workshop*, pages 39–52, 1993.

[4] J.-P. Banâtre, C. Bryce, and D. Le Métayer. An approach to information security in distributed systems. In *Proceedings of the IEEE International Workshop on Future Trends in Distributed Computing Systems*, pages 384–394, 1995.

[5] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 253–267, 2002.

[6] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1984.

[7] D. Bell and L. LaPadula. Secure computer systems: mathematical foundations. *Report MTR 2547 v2, MITRE*, November 1973.

[8] C. Bernardeschi, N. D. Francesco, and G. Lettieri. Using standard verifier to check secure information flow in java bytecode. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference*, pages 850–855, 2002.

[9] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis for secrecy and non-interference in networks of processes. In *PACT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 27–41, 2001.

[10] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the π-calculus with applications to security. *Information and Computation*, 168:68–92, 2001.

[11] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Proceedings of the Second International Conference on The Practical Applications of Java*, pages 63–73, April 2000.

[12] K. Cooper, T. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Available at http://www.cs.rice.edu/ keith/embed., 2001.

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

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

[15] Department of Defense. *Trusted Computer System Evaluation Criteria, DOD standard 5200.28-STD*. 1985.

[16] Ú. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigms Workshop*, pages 87–95, Ontario, Canada, 22–24 1999. ACM SIGSAC, ACM Press.

[17] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[18] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb, 2002.

[19] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 57–65, 1977.

[20] J. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, England, 1973.

[21] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[22] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information flow for virtual machines. In *International Workshop on Programming Language Interference and Dependence*, September 2005.

[23] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, 1998.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international World Wide Web conference*, pages 40–52, 2004.

[25] P. A. Karger, T. E. Leonard, and A. H. Mason. Computer with virtual machine mode and multiple protection rings, U.S. Patent No. 4787031, November 1988.

[26] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[27] J. K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, 1976.

[28] J. K. Millen. Information flow analysis of formal specifications. In *SP '81: Proceedings of the 1981 IEEE Symposium on Security and Privacy*, page 3, 1981.

[29] M. Mizuno and D. A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[30] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[31] A. C. Myers. *Mostly-static decentralized information flow control*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[32] R. Rhode. Secure multilevel virtual computer systems. Technical Report ESD-TR-74-370, MITRE Corp., Bedford, Massachussetts, 1975.

[33] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[34] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, pages 201–220, 2001.

[35] 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. In *37th International Symposium on Microarchitecture*, December 2004.

[36] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 IBM Center for Advanced Studies Conference on Collaborative research*, pages 125–135, 1999.

[37] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW '97: Proceedings of the 10th IEEE Computer Security Foundations Workshop*, page 156, Washington, DC, USA, 1997. IEEE Computer Society.

[38] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[39] C. Weissman. Secure computer operation with virtual machine partitioning. In *National Computer Conference, May 19-22, 1975, Anaheim, California*, pages 929–934, 1975.