# Analyzing Java Bytecode for Secure Information Flow – and Safely Transporting It

Deepak Chandra
Vivek Haldar
Michael Franz
University of California
Irvine, CA 92697
{dchandra,vhaldar,franz}@uci.edu

## ABSTRACT

Static analysis techniques for detecting illegal information flows have great potential, but they are extremely costly – far too costly to perform on-line in a just-in-time compilation context. On the other hand, it is exactly in the context of downloadable machine-independent code that such techniques are the most useful. We were able to reconcile this conflict by performing the information-flow analysis off-line. Proof-carrying code techniques are then used to transport the analysis results, in a manner that can be verified on-line at a relatively modest cost by the code consumer. Our approach operates directly on Java bytecode and can therefore be easily retrofitted even for existing Java programs for which no source code is available.

## 1. INTRODUCTION

High-level language runtimes and virtual machines are becoming increasingly popular platforms for development. More and more code is now being targeted at these language runtimes that execute some form of safe, platform-independent bytecode. The most prevalent examples of this are the Java Virtual Machine (JVM) [1], and the more recent .NET Common Language Runtime (CLR) [2]. Such code platforms offer several advantages over native code. The virtual machine performs a number of static and dynamic checks to ensure a basic level of code safety – type-safety, and control flow safety. Type safety ensures that operators and functions are applied only to operands and arguments of the correct types. A special case of type safety is memory safety, which prevents reading and writing to illegal memory locations – for example, beyond the bounds of an array – and thereby also provides separation between different processes without the need for hardware-based memory management [3]. Control flow safety prevents arbitrary jumps in code (say, into the middle of a procedure, or to an unauthorized routine). These basic properties of safe code are enforced by a combination of static (e.g. bytecode verification) and dynamic (e.g. array bounds checks) techniques. Thus, safe code does away with a major source of errors and vulnerabilities in current systems that stem from unsafe memory operations in C – such as buffer overruns and format string attacks.

While virtual machines provide a portable and safe development target, their current security mechanisms are geared towards discretionary controls only. These are imposed to limit access to critical system resources. However, once access to a resource is gained, there are no mechanisms to track the usage and ensure that it respects the security constraints of the system. Essentially, the entity is *trusted* to handle the resource properly. As a system gets large and complex, it becomes increasingly hard to ensure its trustworthiness. It may be breached because of an unintentional programming error or because of type-safe but malicious code.

In this paper, we present a static analysis to explicitly track the flow of information through a program at a fine-grained level. Our analysis works on Java bytecode, and given security annotations on security-sensitive variables, checks for the presence of illegal flows of information in the program. There are several analyses for checking information-flow properties at the source level on Java [14] and other languages. However, it is not always practical to assume the presence of source code. A lot, if not most, deployed Java code exists as bytecode. Also, the code consumer cannot be certain of the properties checked by such an analysis without actually getting the source and repeating the analysis itself.

Static analysis, however, is too costly to perform when code is dynamically compiled at runtime. To mitigate this, we *safely* transport the results of our analysis. We annotate the analyzed bytecode with the results of the analysis in such a way that they can be easily and quickly verified at the target site before being used. Our analysis is an iterative dataflow analysis, and could, in the worst case, take time quadratic in the size of the program. Thus, providing results in a manner that is easily *verifiable* saves precious time while doing dynamic compilation at the consumer's end. A code producer can now convince a code consumer that the code satisfies certain information-flow properties, while not requiring the consumer to repeat the entire expensive analysis.

To further motivate our approach, consider the following three scenarios:

**Scenario 1 – Downloaded program manipulating sensitive data:** Consider a Java applet from an untrusted (or partially trusted) site that computes tax returns. The program has access to private information such as Social Security Numbers (SSN) and salary information, and it needs to communicate with its "home base" to consult tax tables and to charge the user's credit card for the service. How can we be sure that it isn't also leaking the user's SSN, salary, and bank account information? Clearly, in current Java systems, we cannot.

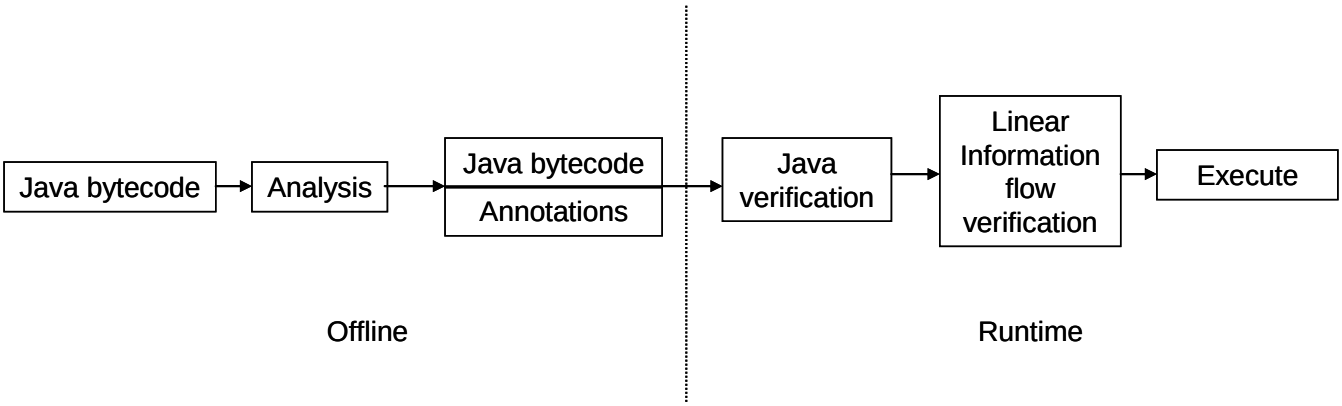


**Figure 1: Overview of our mobile code pipeline**

This scenario highlights a very common problem faced by users today: they use programs that manipulate various sensitive information items on their behalf, and yet, they are given no mechanisms to control how these programs handle their data. They essentially have to trust the program to behave correctly, and not leak secrets to untrusted sources.

**Scenario 2 – Java program handling sensitive databases:** Consider a Java program connecting to two databases, one of which contains sensitive information, and another that contains public information. Even though data from the sensitive database is probably marked as such, once it has been read into a Java program, this meta-information is lost and becomes un-enforceable. Nothing prevents the program from reading rows from the sensitive database, and writing them into the public database. In general, detecting "channels" in Java programs in the current situation requires auditing of source code.

**Scenario 3 – Application server handling sensitive user input:** Consider a web application server that presents a web form for user input. Some of the information is sensitive and hence sent over the wire using an SSL connection. But at both ends, there is no distinction between the sensitive information so secured and the rest of the data – both in the server and the client's browser, this information lives side by side and is potentially vulnerable to programming errors or malicious code. Our information-flow analysis can provide labeling of sensitive data, and separation of such data from non-sensitive data.

Analyzing information-flow properties at the bytecode-level allows us to get some guarantees about a large existing body of code without source-level access, or requiring source-level changes.

The rest of this paper is organized as follows: Section 2 gives an overview of current techniques for access control and information flow control at the source level, and in language runtimes; Section 3 presents our static analysis; Section 4 explains how we safely transport the results of our analysis; Section 5 discusses the status of our implementation and future work; Section 6 presents related work and Section 7 concludes.

## 2. EXISTING SOLUTIONS

Early Java implementations (up to JDK 1.1) had two distinct security environments. The first environment, a complete sandbox, was designed to constrain the execution of applets downloaded from the Web. These applets were considered completely untrusted. The sandbox disallowed any access to the local filesystem, as well as any network connections to domains other than the one from which the applet originated. This sandbox policy was designed to prevent untrusted code from leaking local data, and consuming too many network resources. The second environment had no constraints at all, and was used to run local code on a machine. Code on the local disk was considered completely trusted. Thus, this early model was essentially all-or-nothing, accounting for either completely untrusted or completely trusted code. It had no gradations between these two extremes.

Later versions of Java (after JDK 1.2) added capabilities to create more graded security environments, and provide a variety of more fine-grained security permissions [4]. Instead of being trusted (local), or untrusted (remote), code was now associated with principals. A public key infrastructure and cryptographic signatures were used to bind principals to code. A security policy specified what permissions code originating from various principals would get. Permissions included filesystem read and write permissions, and network socket capabilities. Enforcement was relegated to a runtime security manager that regulated access to privileged resources by looking up the permissions possessed by the object that made the request. For example, a policy may specify that all code digitally signed by the domain `uci.edu` is allowed to read any local file, but to write only under `/tmp`.

However, there are many useful security polices that the current Java architecture does not address. Higher level policies that depend on program state cannot be specified. An example of such a policy is "do not allow transmitting on the network after reading from the local filesystem". Inlined reference monitors [5] and software fault isolation [6] have been used to enforce policies such as this. But even those techniques cannot handle stronger policies that track information within a program. An example of such a policy is: "any data read from the local filesystem must not be transmitted on the network". Note that this is a finer-grained policy than the earlier one because it permits sending on the network even after a local file has been read – it merely forbids sending information that was actually read from the file.

Another shortcoming of the standard Java security architecture is that policies can only be specified in terms of permissions exposed by the Java security API. Another critical drawback is that once a security check is done, there are no controls on the propagation of data thereafter. Data confidentiality

policies cannot be expressed or enforced in the current Java scheme. This is the reason why a policy such as "any data read from the local filesystem must not be transmitted on the network" cannot currently be expressed.

At the Java source level, fields and classes can be marked with access modifiers such as `public`, `private` and `protected` to limit their visibility to other classes and packages. While enforced offline by the Java compiler, marking a field private does not mean that it is inaccessible at runtime. Private fields can easily be accessed using Java's reflection capabilities. Thus, these modifiers should be thought of as an abstraction tool to hide implementation details, rather than as tools for strict protection of information.

Myers' Jflow system [14] uses a Java-based source language, Jif, with security labels on variables. A type system is imposed on these labels, and successful type checking of a program implies the lack of any illicit flows in it. It is uncommon to have access to the source of most deployed programs. The vast majority of these are simply packaged as Java bytecode. Moreover, the lack of an easily checkable proof of the analysis means that the code consumer has to trust the entire Jif compiler.

# 3. ALGORITHM

## 3.1 Background

Variables have associated *security labels* that indicate the sensitivity of the information they hold. We assume that these labels form a lattice.

Information is said to *flow* from *x* to *y* if the value of *y* is derived in any way from the value of *x*.

A *leak* is said to happen when information flows from a variable to another one with a lower security label.

There are two ways a program could leak information. The most direct way is through an *explicit flow* – this is when there is a direct flow of information from one variable to another, for example, through an assignment, or by getting the result of an arithmetic operation.

Another way to leak information is by using control flow. For example, consider the following code snippet, where *x* is secret, and *y* is public:

```
if (x == 0) then y = 1 else y = 2;
print y
```

In this example, even though there is no direct assignment from *x* to *y*, the value of *y* can be easily used to infer the value of *x*. Such flows of information are caused by conditional control flow, and are called *implicit flows.*

## 3.2 Analysis

We present our analysis here for three-address code. To perform this on Java bytecode, we first convert it to three-address code, and then back to Java bytecode. These transformations are explained in greater detail in the next section.

The label of a variable at any point in the flow graph indicates the security level of the value it holds. This is used to prevent explicit flows.

Aside from labels associated with variables, we also attach a security label with each basic block in the control flow graph of our program. We call the former *variable labels*, and the latter *block labels.* Both labels are drawn from the same lattice of security labels.

The block label models the information that is implicitly present at a location in the control flow graph because of conditional jumps on the values of variables. This is used to prevent implicit flows.

We use the following notation: for a variable *x*, $\underline{x}$ denotes its security label. For labels $\underline{x_1}, \underline{x_2}, \ldots \underline{x_n}$, $\underline{x_1} + \underline{x_2} + \ldots + \underline{x_n}$ denotes the least upper bound of those labels.

We assume that methods have been provided with *security signatures* that specify the security labels of its parameters and its return value. For every method, our analysis checks whether the method satisfies its own security signature. While doing this analysis, at method calls it checks if the parameters for the call are compatible with the signature of the method called, and assume that the return value has the label given in the signature. A parameter is compatible if its security label is at most as sensitive as the label specified in the signature. For example, a method taking a "secret" parameter could also take a "public" parameter, but not vice versa.

Our data flow analysis iterates over two kinds of values: variable labels, and block labels. It terminates when they have stabilized. We associate two sets and a block label with each basic block *B* in the control flow graph under consideration:

- IN(B) = $\{x_1{:}l_1, x_2{:}l_2, \ldots, x_n{:}l_n\}$: the *IN* set for the data flow analysis. This is a mapping from variables to their security labels – variable $x_i$ has security label $l_i$. The IN set for a block represents the mapping that holds at the entry of the block.
- OUT(B) = $\{x_1{:}l_1, x_2{:}l_2, \ldots, x_n{:}l_n\}$: the *OUT* set for the data flow analysis. This is also a mapping from variables to their security labels, just like the IN set, but it holds at the exit of a block.
- L(B) = $\{l_1{:}B_1, l_2{:}B_2, \ldots, l_n{:}B_n\}$: This is a list of security labels, along with a pointer to the basic block containing the jump instruction that caused each label to be set. We need this information for declassification. We use the notation $\underline{L(B)}$ to denote the least upper bound of all the labels in L(B).

Initially, the IN and OUT sets for each block are empty, and all variables and blocks are marked with the least-restrictive element of the security-label lattice (*public*). We assume that the formal parameters of the method being analyzed have been labeled with security labels. These labels form the IN set of the entry-block of the control flow graph of the method.

After this initial setup, we perform a standard iterative data-flow analysis which terminates when the IN and OUT sets and block labels have stabilized.

On every iteration of the dataflow analysis, and for every basic block, we perform the following actions:

- **Compute the IN set of this block from its immediate predecessors:** The OUT sets of all the immediate

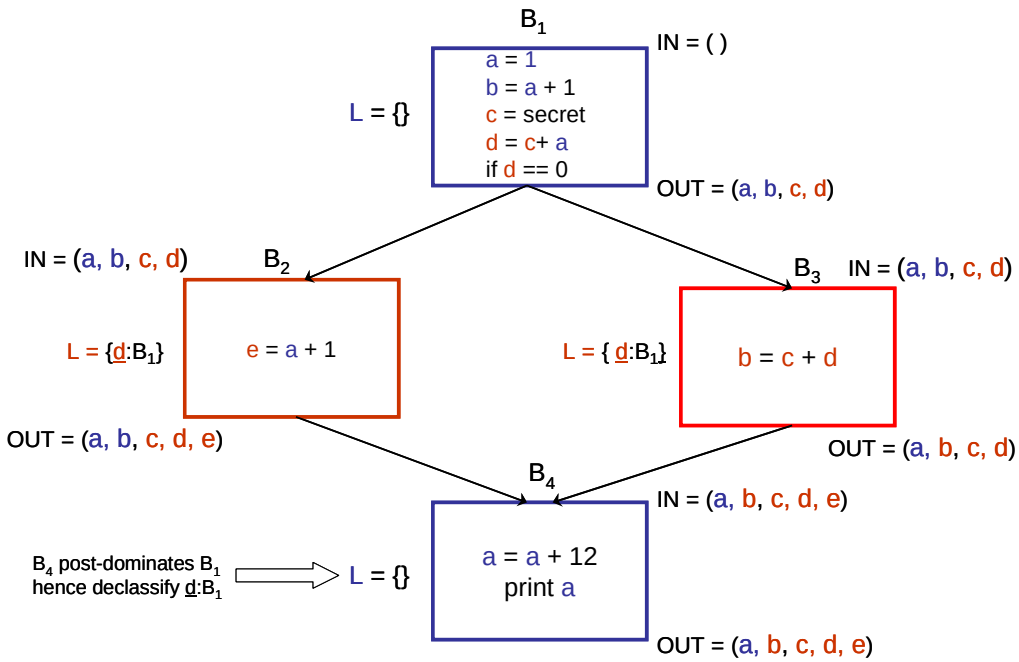


**Figure 2: our information-flow analysis on a sample control flow graph.**

predecessors of the current block are merged to form the IN set of the current block. If a variable *x* occurs in only one of the OUT sets, then it is simply carried over to the IN set. If a variable *x* occurs in multiple OUT sets with labels $l_1, l_2, \ldots, l_n$, then it is put in the IN set with label $l_1 + l_2 + \ldots + l_n$.

- **Compute the block label of this block from the block labels of its immediate predecessors and post-dominator information *(this is where we explain declassification)*:** The block label of the current block label is computed as follows: it is simply the union of the block labels of all the immediate predecessors of the current block, except when the same label *l:B* occurs in multiple predecessor block labels. In that case, we check if the current block post-dominates B. (A block $B_1$ post-dominates another block $B_2$ if every path from $B_2$ to the exit of the control flow graph goes through $B_1$). If it does, then we can safely declassify the security label *l* for this block, and remove it from the current block label.

- **Propagate labels through the block and compute the OUT set:** At the beginning of a block, the OUT set for that block is initialized with the IN set. We go through the basic block instruction by instruction, and for each instruction *"x = y op z"* in the block, we compute the label of *x* as follows: $\underline{x} = \underline{y} + \underline{z} + \underline{L(B)}$. The labels $\underline{y}$ and $\underline{z}$ are looked up in the OUT set. Finally, we are left with a mapping from variables to their respective security labels. This mapping holds at the exit of the block. This is the OUT set for that block.

Figure 2 gives an example of our analysis performed on a simple control flow graph. At the entry block, the variables *c* and *d* are marked "secret", and everything else is "public". The example shows how other variables get tainted secret, and how implicit flows are modeled using block labels.

We now give a brief sketch of the proof of correctness of our analysis. Given that initially the method's formal parameters are explicitly labeled with security labels, all we need to show is that every variable to which a value of label *l* flows is also marked with a label at least as sensitive as *l*. This is straightforward: for the case of explicit flows, we can easily see (by construction in our analysis) that the left-hand sides of expressions are labeled the least upper bound of variables on the right-hand side *and* the block label. For the case of implicit flows, we elevate the label of a block if control flows to it as a result of a conditional jump, and henceforth every value computed in that block is also labeled at least as sensitive as its block label. Hence, we see that security labels are faithfully propagated by our analysis.

## 4. TRANSPORTING RESULTS SAFELY

We now explain how we annotate Java bytecode with an easily and quickly (in time linear to program size) verifiable proof that it satisfies the information flow properties that our analysis checked for.

We annotate Java bytecode with the following:

- The bytecode for each method is annotated with where its basic block boundaries lie. We assign a unique number to each block.

- Each block is then annotated with the final stabilized values of it's IN and OUT sets, and its block label.

- Each method is annotated with the post-dominator tree for its control-flow graph.

Given these annotations, we now explain how they can be quickly verified in one pass over the bytecode.

For each block B, we first verify that it's IN set is correct. For the entry set, the IN set is directly derived from the security labels associated with formal parameters, so we simply check that the IN set of the entry block is compatible with the labels of parameters. For all other blocks, we compute a new IN set for the block from the OUT sets of all its immediate predecessors. This is done in exactly the same way as is done by the analysis. Finally, we check that the annotated IN set is stricter than the newly computed IN set. Similarly, we also compute a new block label of this block in the same way as in the analysis. Then we check than the annotated block label is stricter than the newly computed block label.

To verify the OUT set of a block B, we compute a new OUT set (starting with the annotated IN set) by sequentially going through the instructions in the block. Then we check that the annotated OUT set is stricter then the newly computed OUT set.

To compute the block labels, we need post-dominator information. We transport this is an annotation for every method. To verify that the post-dominator information is correct, we use the technique of [16], which given a control flow graph and a post-dominator tree, verifies in linear time that the post-dominator tree is correct.

We now give a sketch of the proof of correctness of the above verification procedure. We need to show that a malicious adversary cannot modify the annotations in a way that can violate the property proved by the static analysis:

(1) every variable to which a value of label *l* flows is also marked with a label at least as sensitive as *l*.

Essentially, we want to show that if an adversary "lies" about an annotation, it will be caught by our verification procedure. For the entry block of the procedure, it's IN set can be trivially checked as it should have labels at least as sensitive as those of the formal parameters of the method under consideration. We call an annotated IN set of a block *correct* if the label of every variable in it is at least as strict as that computed by taking the unions of the OUT sets of all the immediate predecessors of the block. Similarly, an annotated OUT set is *correct* if every variable in it is at least as sensitive as that in the OUT set as computed by the analysis. For our annotations to be incorrect there must be at least one block for which either the IN or OUT set is incorrect. This is because if the IN and OUT sets of every block are correct, and we can trivially check the correctness of the IN set of the entry block, then by the correctness of our static analysis algorithm, this program does not violate (1).

# 5. STATUS AND FUTURE WORK

We have currently implemented the analysis framework. We have based our implementation on the Soot framework [27]. We use Soot to parse Java bytecode into Soot's Jimple representation. Jimple is a conversion of Java bytecode to three-address code. We perform our analysis on Jimple.

The proof generator and verifier are currently under development. We will need to "translate" our proof from Jimple back to Java bytecode in order to transport it. This can be done by simply converting all temporaries in three-address Jimple to local variables in the final bytecode.

There are several other broad avenues of future work we would like to consider.

Another approach to tackling the information-flow problem is to use dynamic techniques, such as mandatory access [28]. Such approaches have one very significant advantage over static techniques: they can be bound to code very late – at execution time. This gives them a great deal of flexibility. For example, different executions of the same program could be run under varying security policies.

The disadvantage of dynamic methods when compared to static techniques is that they lack a clean mechanism to safely declassify information. This is because dynamic techniques only know about one execution path, whereas static methods can reason about all execution paths taken to reach a certain program point. This leads to "label creep", where security labels on objects keep moving up, often making programs unusable.

We have also designed and implemented a system that does mandatory access control in the Java virtual machine at the level of objects [28]. We now want to investigate how this dynamic approach and the static approach presented in this paper can be meaningfully combined to get the best features of each. For example, we could annotate bytecode with verifiable information about points where it is safe to declassify information. Such a hybrid technique will have all the advantages of dynamic techniques, but will also be able to make informed decisions at runtime about where it is safe to declassify certain information.

Currently, we expect the user to provide security labels the formal parameters of methods. This is a very cumbersome and error-prone approach. We would like to find a way in which the user could specify tags in a high-level manner. For example, one could specify tags at a package level. So, all method parameters of classes in `java.io` are public, or all classes in `com.taxform` are secret, except the ones explicitly marked public.

All current approaches to securing information flow terminate the program (in the case of dynamic techniques), or reject a program (in the case of static techniques) when they find that the program leaks information. This behavior is of little use to the user or programmer because she doesn't know either what particular piece of information is getting leaked, or where it's getting leaked. Could we try to identify the source of the information getting leaked? For example, could we inform the user that the information getting leaked is from the file /etc/passwd? Such information would be useful in making the decision of whether to let the program continue running, or terminate it right away.

Another useful piece of information to report would be not only what information is getting leaked, but *how much* of it is getting leaked. For example, even legitimate programs leak information about passwords when they check whether a user-provided password matches the correct password. Quantifying leaks is also useful for specifying high-level policies such as saying that a program is safe as long as it only leaks 1/n bits per execution.

**Covert channels:** Explicit channels for the transfer of information, such as assignments or method calls, can be controlled by changing or monitoring the mechanisms that implement them. However, information can also be transmitted

through covert channels that do not depend on explicit mechanisms, but the side-effects of computation [7]. Examples are:

- **Timing channels:** measuring how long a computation took can reveal something about the data it was operating on [8]. A subset of timing channels are termination channels, where the termination of a program reveals information [9].
- **Power channels:** measuring the power consumption of a CPU (or a peripheral, such as a smart card) can be used to infer the bits being computed.
- **Resource channels:** information could be leaked by monitoring the consumption (or exhaustion) or various resources, such as a CPU or memory.

A full-fledged Java virtual machine has many potential covert channels. Examples include: how often, when and how long the garbage collector runs. For example, code could be crafted to purposely trigger the garbage collector. Measuring the latency of garbage collection could reveal in-formation about the size and number of objects. This is an instance of using resource consumption as a covert channel. While our technique can control the overt flow of information in Java bytecode, stemming the flow through covert channels remains an open question.

Note, however, that a virtual execution environment can randomize a number of its activities to stem the flow of information from covert channels. For example, it could insert random delays into the execution of individual bytecode instructions. It could even randomize its heap allocation and garbage collection to defeat attacks that observe the heap or garbage collection delays. Such randomization is easier to do in a virtual machine, as opposed to native code running on bare hardware. This is because of the higher-level nature of a virtual machine, and because all code is executed under complete control of the virtual machine.

# 6. RELATED WORK

Early work in information flow and mandatory access control (MAC) was done by Bell and LaPadula [10], who pioneered the idea of information being classified at multiple sensitivity levels. Denning [11] extended the Bell-LaPaulda model to use a lattice for sensitivity labels. Denning was also one of the first to use static analysis on source code to enforce information flow properties with very little runtime overhead [12]. Volpano formalized the soundness of the analysis that Denning proposed [13]. Andrew Myers et al [14] used a type system to enforce information flow statically. Their Jif compiler is a source-to-source compiler that checks a Java program with information flow annotations, type-checks it, and outputs a regular Java program. This work is closest to ours. However, our analysis is significantly different as we analyze Java bytecode (and not Java source programs) for information flow properties. This enables us to analyze and check commercial off-the-self Java programs and legacy code. Another significant difference is that we can annotate bytecode with a compact, easily-verifiable *proof* of the results of our static analysis. This will enable code consumers to easily check the validity of the information flow properties of the code without repeating the entire (computationally-intensive) analysis.

RIFLE [15] is a system that tracks information flow dynamically. This is accomplished by 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. This converts implicit flows to explicit flows that can then be tracked by the architecture.

The major difference between RIFLE and our system is that our solution is software-only and does not require modifications to the underlying hardware architecture. However, since RIFLE analyses native binaries, it can enforce its constraints on a much wider range of programs, whereas our solution only works for Java bytecode.

Bernardeschi and et.al [26] use type-based abstract interpretation, which is very similar to bytecode verification, to prove information flow safety of java bytecode. They, like Denning, handle implicit flows and use immediate post-dominator to reduce the security label of the execution context. Our approach for analyzing the program for information flow safety is similar. We, unlike them, also transport the result of our analysis in ways that is easily checkable by the code consumer — eliminating the need to do expensive analysis every time before execution of the program.

Much work has been done to annotate Java bytecode in order to reduce the overhead of optimization and code-generation at the consumer's end. Krintz and Calder [17] add various annotations to Java class files with the goal of reducing compilation overhead in a dynamically optimizing compiler. Some of the annotations are: counts of local variable uses, to aid the register allocator; indicating which methods are likely to get inlined multiple times, to save some initial work that is done for every inlining; and method priority annotations, to guide the dynamic optimizer's choice of which methods to optimize more aggressively. In [18], Jones and Kamin present a scheme for annotating Java class files with virtual register assignments in order to facilitate register allocation. The annotation-aware JIT (AJIT) system [19] also adds annotations to Java class files that are the results of various time-consuming optimizations. They add annotations for null-check elimination, array bound check elimination, virtual register assignment, and memory disambiguation.

Note that these are optimizations that do not affect program semantics, only performance. Their common feature is that they serve as "effort-directing" information for a just-in-time (JIT) compiler. This allows them to be used safely at the code consumer without verifying them first.

Stork et. al have used a mobile code format based on abstract syntax trees (ASTs) [21] as a base for safe transportation of annotations. In [22], they show how to annotate ASTs with the results of escape analysis, and then verify them. The major difference is the intermediate format to which these annotations are added. They add annotations to a mobile code format based on abstract syntax trees, which is a much higher level representation than the bytecodes we have considered here.

In broad terms, our work is a very specific case of the much more general concept of proof-carrying code (PCC) [20]. Like PCC, a computation-intensive "proof-generator" emits a small certificate (the annotations) that is verified at the code consumer's end in one linear pass. Also like PCC, the verifier, which is small itself, does not need to trust the program that generates annota-

tions. This keeps the trusted computing base small. However, the full generality of PCC was not needed in this case, where a simple, specific solution existed.

The work that comes closest to that presented here is stackmaps [23], as used in Java 2 Micro Edition(J2ME). Stackmaps allow the costly Java bytecode verification procedure to be split into two phases – a costly off-device pre-verification phase that augments class files with stackmaps, and a quick on-device verification phase that checks the stackmaps in one linear pass. Also called "lightweight verification" stackmaps were first proposed in [24], and further formalized in [25]. The Java bytecode verification procedure, though not an optimization, is also an iterative data flow analysis. Stackmaps are also essentially "frozen" data flow results, as are our annotations.

# 7. CONCLUSION

Access control mechanisms in current language runtimes lack support for expressing and enforcing fine-grained information-flow policies. Such policies are needed to enforce the correct handling of sensitive data. As a solution, we present a static analysis on Java bytecode that certifies its information-flow properties. Moreover, to make checking information-flow viable in a JIT-compiled environment, we also present a scheme for annotating Java bytecode with the results of our analysis in way that is easily and quickly verifiable at the target site.

# 8. ACKNOWLEDGEMENTS

Parts of this effort are sponsored by the National Science Foundation under grants CCR-TC-0209163 and CCR-ITR-0205712, and by a generous gift from Sun Microsystems Labs.

Any opinions, findings, and conclusions or recommendation expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), any other agency of the U.S. Government, or those of Sun Microsystems, Inc.

# 9. REFERENCES

[1] Tim Lindholm and Frank Yellin; The Java Virtual Machine Specification; Addison-Wesley, April 1999.

[2] Microsoft Corporation. Microsoft .NET, 2003.

[3] B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In Proceedings of the 5th Workshop on Hot Topics in Operating Systems, pages 62–65, Orcas Island, WA, May 1995.

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

[5] Úlfar Erlingsson, Fred Schneider; IRM Enforcement of Java Stack Inspection; IEEE Symposium on Security and Privacy, 2000.

[6] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203–216, 1993.

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

[8] J. Agat. Transforming out timing leaks. In ACM Symposium on Principles of Programming Languages, pages 40–53, 2000.

[9] Volpano and Smith. Eliminating covert flows with minimum typings. In The 10th Computer Security Foundations Workshop. IEEE Computer Society Press, 1997.

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

[11] D. E. Denning. The lattice model of secure information flow. Commun. ACM, 19(5):236–243.

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

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

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

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

[16] Loukas Georgiadis and Robert E. Tarjan. Dominator Tree Verification and Vertex-Disjoint Paths in Digraphs. In ACM-SIAM Symposium on Discrete Algorithms, 2005.

[17] C. Krintz and B. Calder. Using annotation to reduce dynamic optimization time. In SIGPLAN Conference on Programming Language Design and Implementation, 2001.

[18] J. Jones and S. Kamin. Annotating Java class files with virtual registers for performance. Concurrency: Practice and Experience, 12(6), 2000.

[19] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating Java bytecodes in support of optimization. Concurrency: Practice and Experience, 9(11), 1997.

[20] G. C. Necula. Proof-carrying code. In The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1997.

[21] V. Haldar, C. H. Stork, and M. Franz. The source is the proof. In New Security Paradigms Workshop, 2002.

[22] M. Franz, C. Krintz, V. Haldar, and C. H. Stork. Tamper-proof annotations, by design. Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, 2002.

[23] Sun Microsystems Inc. Connected, limited device configuration, Apr 2000.

[24] E. Rose and K. H. Rose. Lightweight bytecode verification. In Workshop on Formal Underpinnings of the Java Paradigm (OOPSLA 1998), 1998.

[25] G. Klein and T. Nipkow. Verified lightweight bytecode verification. Concurrency: Practice and Experience, (13), 2001.

[26] Marco Avvenuti and Cinzia Bernardeschi and Nicoletta De Francesco. Java bytecode verification for secure information flow. SIGPLAN Notices, 38(12), 2003.

[27] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon and Phong Co. Soot – a Java optimization framework. In proceedings of CASCON, 1999.

[28] Vivek Haldar, Deepak Chandra and Michael Franz. Practical, Dynamic Information-Flow for Virtual Machines. Submitted for publication.