Enforcing Java Run-Time Properties Using Bytecode Rewriting

Algis Rudys and Dan S. Wallach

Rice University, Houston, TX 77005, USA
(arudys|dwallach)@cs.rice.edu

Abstract. Bytecode rewriting is a portable way of altering Java's behavior by changing Java classes themselves as they are loaded. This mechanism allows us to modify the semantics of Java while making no changes to the Java virtual machine itself. While this gives us portability and power, there are numerous pitfalls, mostly stemming from the limitations imposed upon Java bytecode by the Java virtual machine. We reflect on our experience building three security systems with bytecode rewriting, presenting observations on where we succeeded and failed, as well as observing areas where future JVMs might present improved interfaces to Java bytecode rewriting systems.

1 Introduction

Bytecode rewriting presents the opportunity to change the execution semantics of Java programs. A wide range of possible applications have been discussed in the literature, ranging from the addition of performance counters, to the support of orthogonal persistence, agent migration, and new security semantics. Perhaps the strongest argument in favor of bytecode rewriting is its portability: changes made exclusively at the bytecode level can be moved with little effort from one Java virtual machine (JVM) to another, so long as the transformed code still complies to the JVM specification [1]. An additional benefit is that code added by bytecode rewriting can still be optimized by the underlying JVM.

JVMs load Java classes from disk or elsewhere through "class loaders," invoked as part of Java's dynamic linking mechanism. Bytecode rewriting is typically implemented either by statically rewriting Java classes to disk, or through dynamically rewriting classes as they are requested by a class loader. This process is illustrated in Figure 1.

In this paper, we describe three systems which we have built that use bytecode rewriting to add security semantics to the JVM. SAFKASI [2] is a bytecode rewriting-based implementation of stack inspection by security-passing style. Soft termination [3] is a system for safely terminating Java codelets.[1] Finally, transactional rollback [4] is a system for undoing the side-effects of a terminated codelet, leaving the system in a consistent state suitable for, among other thing, restarting terminated codelets.

[1] The term "codelet" is also used in artificial intelligence, numerical processing, XML tag processing, and PDA software, all with slightly different meanings. When we say "codelet," we refer to a small program meant to be executed in conjunction with or as an internal component of a larger program.

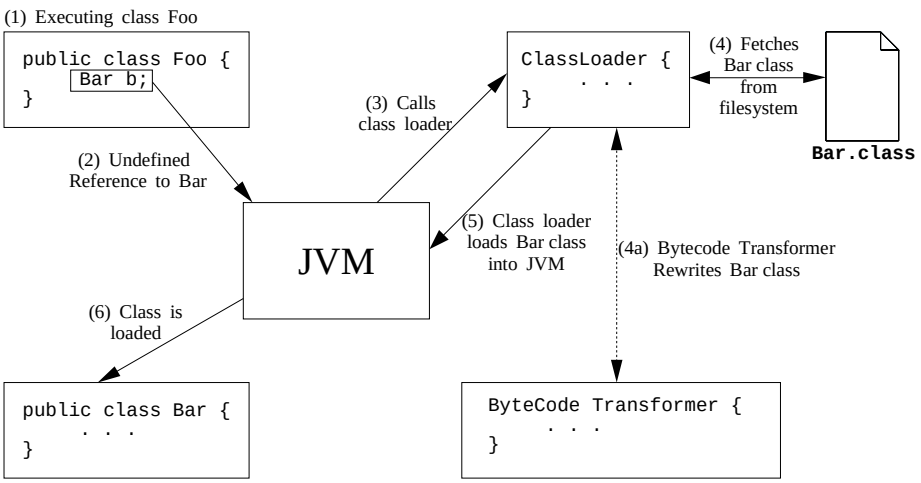

Fig. 1. How a Java bytecode transformation changes the process of loading a class. If an already loaded class, `Foo`, uses an as yet undefined class `Bar` (either accesses a static member or creates an instance) (1), the JVM traps the undefined reference to `Bar` (2), and sends a request for the class loader to load the class (3). The class loader fetches the class file (`Bar.class`) from the filesystem (4). In standard Java, the input class is then loaded into the JVM (5). In a bytecode rewriting system, the bytecode transformer is first invoked to transform the class (4a). In either case, the class is now loaded in the JVM (6).

Java bytecode rewriting has been applied in far too many other systems to provide a comprehensive list here. We cite related projects in order to discuss the breadth of the use of the technique.

Access Control. By intercepting or wrapping calls to potentially dangerous Java methods, systems by Pandey and Hashii [5], Erlingsson and Schneider [6], and Chander et al. [7] can apply desired security policies to arbitrary codelets without requiring these policies to be built directly into the Java system code, as done with Java's built-in security system.

Resource Management and Accounting. J-Kernel [8] and J-SEAL2 [9] both focus primarily on isolation of codelets. Bytecode rewriting is used to prevent codelets from interfering in each others' operations. JRes [10] focuses more on resource accounting; bytecode rewriting is used to instrument memory allocation and object finalization sites.

Optimization. Cream [11] and BLOAT (Bytecode-Level Optimization and Analysis Tool) [12] are examples of systems which employ Java bytecode rewriting for the purpose of optimization. Cream uses side-effect analysis, and performs a number of standard optimizations, including dead code elimination and loop-invariant code motion. BLOAT uses Static Single Assignment form (SSA) [13] to implement these and several other optimizations.

Profiling. BIT (Bytecode Instrumenting Tool) [14] is a system which allows the user to build Java instrumenting tools. The instrumentation itself is done via bytecode rewriting. Other generic bytecode transformation frameworks, such as JOIE [15] and Soot [16], also have hooks to instrument Java code for profiling.

Other Semantics. Sakamoto et al. [17] describe a system for thread migration implemented using bytecode rewriting. Marquez et al. [18] describe a persistent system implemented in Java entirely using bytecode transformations at class load time. Notably, Marquez et al. also describe a framework for automatically applying bytecode transformations, although the status of this framework is unclear. Kava [19] is a reflective extension to Java. That is, it allows for run-time modification and dynamic execution of Java classes and methods.

All of these systems could also be implemented with customized JVMs (and many such customized JVMs have been built). Of course, fully custom JVMs can outperform JVMs with semantics "bolted on" via bytecode rewriting because changes can be made to layers of the system that are not exposed to the bytecode, such as how methods are dispatched, or how memory is laid out. The price of building custom JVMs is the loss of portability.

Code rewriting techniques apply equally to other languages. One of the earliest implementations of code rewriting was Informer [20], which, to provide security guarantees, applied transformations to modules written in any language and running in kernel space. In particular, the transformations discussed in this paper could be applied to add similar semantics to other type-safe languages like Lisp and ML as well as such typed intermediate representations as Microsoft's Common Language Infrastructure and typed assembly languages.

A number of issues arise in the course of implementing a bytecode rewriting system. In this paper, we describe our experiences in implementing three such system in Section 2. Section 3 discusses JVM design issues that we encountered when building our systems. Section 4 discusses optimizations we used to improve the performance impact of our systems. Finally, we present our conclusions in Section 5.

2 Bytecode Rewriting Implementations

This paper reflects on lessons learned in the implementation of three security systems built with Java bytecode rewriting. The first, SAFKASI [2], uses bytecode rewriting to transform a program into a style where security context information is passed as an argument to every method invocation. Soft termination [3] is a system for safely terminating Java codelets by trapping backward branches and other conditions that might cause a codelet to loop indefinitely. Transactional rollback [4], intended to be used in conjunction with soft termination, allows the system to undo any side-effects made to the system's state as a result of the codelet's execution, returning the system to a known stable state.

2.1 SAFKASI

SAFKASI [2] (the security architecture formerly known as stack inspection), is an implementation of Java's stack inspection architecture [21] using Java bytecode rewriting. SAFKASI is based on *security-passing style*, a redesign of stack-inspection which provably maintains the security properties of stack-inspection, improves optimizability and asymptotic complexity, and can be reasoned about using standard belief logics.

Stack inspection is a mechanism for performing access control on security-sensitive operations. Each stack frame is annotated with the security context of that stack frame. Stack inspection checks for privilege to perform a sensitive operations by inspecting the call stack of the caller. Starting at the caller and walking down, if it first reaches a frame that does not have permission to perform the operation, it indicates failure. If it first reaches a frame that has permission to perform the operation and has explicitly granted permission, the operation is allowed. Otherwise, the default action is taken as defined by the JVM.

With security-passing style, instead of storing the security context in the stack, the security context is passed as an additional parameter to all methods. This optimizes security checks by avoiding the linear cost of iterating over the call stack.

SAFKASI is implemented by passing an additional parameter to every method in the system. This parameter is the security context. It is modified by the *says* operator, which is used by a class to explicitly grant its permission for some future operation. The stack-inspection check simply checks this context for the appropriate permission.

Performance. SAFKASI was tested using the NaturalBridge BulletTrain Java Compiler, which compiles Java source to native binary code [22]. With CPU-bound benchmarks, SAFKASI-transformed programs executed 15 to 30% slower than equivalent stack inspecting programs.

2.2 Soft Termination

Soft termination [3] is a technique for safely terminating codelets. The basis for soft termination is that the codelet doesn't need to terminate immediately, as long as it is guaranteed to eventually terminate. We implemented soft termination by first adding a termination flag field to each class. We then instrumented each method, preceding all backward branches with termination checks to prevent infinite loops and beginning all methods with termination checks to prevent infinite recursion.

The termination check simply checks the class's termination flag. If set, a Java exception is thrown. The codelet is free to catch the exception, and resume. However, the next time a backward branch or method call is encountered, the exception is thrown anew. Using this mechanism, we can prove that the codelet is guaranteed to terminate in finite time.

Soft termination distinguishes between system code and user code (that is, codelets). System code should not be interrupted, even if the associated codelet has been marked for termination. Codelets are rewritten by the transformer, while system code is not touched. This result can be achieved by performing the transformation in the Java class

loader. Since the system classes and codelets are naturally loaded by different class loaders, the separation becomes natural.

Blocking calls are also addressed in our system. Blocking calls are method calls, most commonly input/output calls, which wait for a response before returning. We wanted to guarantee that a codelet could not use blocking calls to bypass termination. The `Thread.interrupt()` method allows us to force blocking method calls to return.

To determine which threads to interrupt, we wrap blocking calls with code to let the soft termination system know that a particular thread is entering or leaving a blocking call. This code also uses Java's stack inspection primitives to determine whether the code is blocking on behalf of a codelet or on behalf of the system code. Only threads blocking on a codelet's behalf are interrupted.

Performance We implemented this system and measured the performance when running on Sun Microsystems Java 2, version 1.2.1 build 4. The worst results, of a simple infinite loop, was 100% overhead. In the real-world applications tested, the overhead was up to 25% for loop-intensive benchmarks, and up to 7% for the less loop-intensive benchmarks.

2.3 Transactional Rollback

Transactional rollback [4] was designed to complement soft termination in a resource management system. Soft termination guarantees that system state is never left inconsistent by a codelet's untimely termination. However, the same guarantee is not made about a codelet's persistent state. Any inconsistencies in this state could destabilize other running codelets as well as complicate restarting a terminated codelet.

We solve this problem by keeping track of all changes made by a codelet, and if the codelet is terminated, the changes are undone. Each codelet is run in the context of a transaction to avoid data conflicts. We implement transactional rollback by first duplicating each method in all classes (including system classes). The duplicate methods take an additional argument, the current transaction.

In the duplicate methods, all field and array accesses (that is, field gets and puts and array loads and stores) are preceded with lock requests by the transaction on the appropriate object. Method calls are also rewritten to pass the transaction parameter along. A number of fields are added to a class to maintain a class's backups and lock state for the class.

If the code is not running in a transaction, the methods called are for the most part exactly the original methods. This allows us to limit the performance impact to code running in a transaction.

Performance We implemented this system and measured the performance when running on Sun Microsystems Java 2, version 1.3 for Linux. The overhead of the transaction system ranged from 6 to 23×. The major component in this overhead was in managing array locks.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.