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.

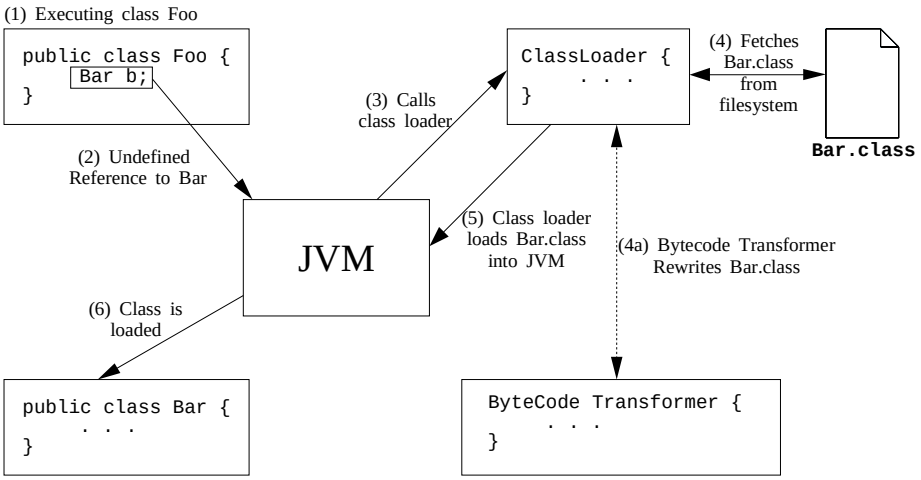

Fig. 1. How a Java bytecode transformation changes the process of loading a class. If an already loaded class, `Foo`, uses an as yet undefined class `Bar` (either accesses a static member or creates an instance) (1), the JVM traps the undefined reference to `Bar` (2), and sends a request for the class 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$\times$. The major component in this overhead was in managing array locks.

3 JVM Design Issues

Bytecode rewriting is inherently different from language run-time system customization. Bytecode rewriting results in classes that are run inside a standard Java virtual machine. These classes treat the JVM as a black box; functionality internal to the JVM cannot be altered. Rewritten classes must be valid bytecode, according to the virtual machine specification [1]. However, because codelets might be malicious, we cannot assume that the code we receive as input was generated by valid Java source, and so cannot necessarily use the Java Language Specification [23] for guidance.

Table 1 summarizes our bytecode rewriting implementations and associated wish list items.

3.1 JVM Internals

The basic distinction of bytecode rewriting as opposed to language run-time system customization is that with bytecode rewriting, operating on JVM internals is not possible.

Class Reloading Once classes are loaded into the JVM, they can neither be unloaded nor reloaded. Likewise, one cannot control when a class's static initializer will be called or when dead objects' finalizers will be invoked. Java version 1.4 includes a feature in its debugger architecture called *HotSwap*, which allows classes to be reloaded[2]. Existing activation records continue to run with code from the original version of the class. Also, the new class's static initializer is not run again, and new fields added to existing classes will be null or zero. This new feature was not available when we were building our systems, and would have been a welcome addition, despite its limitations.

Memory Management One of the features on our "wishlist" is the addition of hooks into the JVM's memory system. Had we been able to exploit the garbage collector's safe point traps, our soft termination system could have performed periodic checks with more flexibility and lower overhead than trapping backward branches. Also, by modifying the garbage collector, we might have been able to enforce memory usage policies that are more robust than those used in JRes [10] and J-SEAL2 [9], which assume that whoever allocated a block of memory should be responsible for "paying" for that memory. We implemented exactly such a mechanism to account for memory usage in IBM's RVM [24] by modifying the RVM's garbage collector [25]. New pluggable garbage collection systems for Java, such as GCTk [26], may allow us to implement such features without requiring changes to the underlying JVM. In addition, the Real-time Specification for Java[3] allows programs to create individual memory regions treated differently by the garbage collector and select from which region memory is allocated.

[2] See `http://java.sun.com/j2se/1.4/docs/guide/jpda/enhancements.html` for more information.

[3] See `http://www.rtj.org/` for more information.

	SAFKASI	**Soft Termination**	**Transactional Rollback**
Added Fields	• Security principal	• Termination flag	• Lock state • Class backup
Modified Methods	• Added security context parameter to each method • Added security context parameter to all method invocations • Add Security-passing style security checks • Added stub methods for upcalls	• Added safe point-like termination checks on backward branches and method calls • Wrapped blocking calls to register with soft termination system to allow interrupting	• Duplicated each method • Added transaction parameter to each duplicated method • Added transaction parameter to each method invocation from duplicated methods • Instrumented field and array accesses within methods • Added code in original methods to detect transactional context
Added Classes		• Wrapping subclass for class with blocking native methods	• Added classes for static and instance backups of each existing class
JVM Issues	• Closed world • System classes • Native methods • Bootstrapping	• Synchronization • Blocking calls	• System classes • Native methods • Bootstrapping • Synchronization • Arrays
Wish List Items	• Class file reloading	• Synchronization interface • Garbage collector interface	• Synchronization interface • Access to array implementation

Table 1. This table summarizes the three systems based on bytecode rewriting. It describes the transformations to Java bytecode, JVM-related issues, and items on the JVM wish list inspired by that particular project.

Threads Thread scheduling is another black box subsystem of the Java virtual machine. No mechanism is provided for either replacing the thread scheduler or fine-tuning how threads are scheduled; the only interface provided is the `Thread.setPriority()` method. The Real-time Specification for Java does allow for replacing the thread scheduler, but has only recently been finalized.

Native Methods Native methods, while not strictly part of the JVM, are also treated as black boxes. We cannot control where a native method might go, and how that native method might behave. Native methods might perform arbitrary computations and are not necessarily guaranteed to return in a timely fashion (e.g., I/O routines might block indefinitely while they wait for data to arrive).

Furthermore, we have no control over up-calls from native methods back to the Java classes which we do control. In particular, we have no access to the Java Native Interface (JNI) calls used by native methods to interface with Java. If we could intercept these calls, then we could transform native methods to see a view of the Java classes consistent with what the Java classes themselves see after being transformed. Since that is not an option with current JVMs, we have adopted a number of strategies to cope with native methods, described in Section 3.3.

Note that certain JVMs implemented in Java, such as BulletTrain and IBM's RVM [24], use fewer native methods than JVMs implemented in machine code. The tradeoff is that they have more methods which are not native but are treated specially by the JVM. In our bytecode transformations, these methods need to be treated the same as standard native methods. See Section 3.3 for details on such special methods and how we handled them.

3.2 "Local" Java Semantics

The flexibility of Java bytecode is both its benefit and its curse. A bytecode rewriting system must deal with every feature in the bytecode. We first deal with "local" features, which is to say, aspects of the Java bytecode where an individual class or method can be rewritten without needing any knowledge of how other classes in the JVM might interoperate with it.

Constructors Constructors have two properties which set them apart from normal methods. First, no operations, whether initialization or otherwise, can be performed until either the super-class constructor or a same-class constructor has been called, although arguments to a super-class constructor may be legally computed. This presented a problem in the soft termination system.

Soft termination is designed to check the value of the termination flag at the beginning of every method, including constructors, but inserting this check into a constructor results in a verifier error. Instead, the best we can do is to check for termination immediately following the call to the super-class constructor. As a result, we may not be able to terminate a loop constructed with constructors that recursively invoke other constructors before calling their super-class constructor.

Transactional rollback requires every object to be locked before it can be modified. Normally, this is accomplished with the addition of a specialized lock instance to every object instance. However, while an object is being constructed, this lock instance will be null. To address this, we needed to add checks *everywhere* an object is accessed to check whether its lock instance is null, slowing down all locking code to handle this special case.

Exceptions Exceptions (and errors) can introduce implicit or asynchronous control flows into a Java method. Asynchronous exceptions, induced by calls to `Thread.stop()`, can occur literally anywhere, and at any time. For this reason, among other reasons, Sun has deprecated the method. Implicit exceptions typically occur as a side-effect of a program error, such as a `NullPointerException` or the various arithmetic errors. Here, the control flow edge from the potentially faulty operation to the construction and throwing of an exception does not appear explicitly in the Java bytecode.

This caused problems for SAFKASI, which needs to pass its security context as a parameter to the exception constructors. The only available choices were to either save the security context in thread-local storage or to give up and pass no context. Using thread-local storage is expensive, requiring a hash table operation on every store, and the exception constructors for Java's implicit exceptions appear to never perform security critical operations. As a result, SAFKASI gives up, allowing these constructors to be invoked with no security context, and letting them start over as if there was no call stack before them.

We also observe that, in Java bytecode, an exception handling block (that is, a `try-catch` block) is represented as offsets delimiting the `try` block and the offset for the start of the `catch` block. The JVM specification requires that the `catch` block start strictly after the `try` block ends. That is, an exception-handler cannot handle its own exceptions. However, Sun's JVM does not enforce this restriction.

This can be used by a malicious codelet to evade a soft termination signal. The exception handler acts as a backward-branch; when an exception is thrown in the exception handler, control passes to an earlier point in the text of the program. If this exception always occurs, like the termination signal of soft termination, the result will be an infinite loop. We solved the problem by having the soft termination system detect and reject such exception-handling blocks.

Arrays Transactional rollback needs to be able to save backup copies of all objects before they are written. For most classes, we can create "backup" fields for every original field in the class. Assigning the backup to refer to the original object is sufficient to preserve the original value of the backup. However, for arrays, this no longer works; there is no place in the array to store a reference to the array's backup. Our solution is to maintain a global hash table that maps arrays to their backups. For each array, the backup array must be the same size as the original. Creating this backup requires copying the whole array. For codelets that make extensive use of arrays, whether large or small, this creates a significant overhead.

Our preferred solution would be for Java to have a mechanism to let us add our own field to all arrays. Java's arrays already track their length; we want one more reference that we can use transparently to the Java application.

Threads Threads can be thought of as a special case of native methods which make up-calls to Java code (in this case, during thread initialization, the up-call happens to be on a new thread while control returns to the original thread). In both SAFKASI and transactional rollback, we needed state computed in the parent thread to be sent to the child thread. This was performed by modifying `java.lang.Thread` to have an additional field where the parent thread can store context information in the child thread. This context is consulted when the child's `run()` method is invoked by the thread run-time system. Since threads are implemented in Java, this modification can be performed as part of the bytecode-rewriting process.

Another important issue with threads is controlling them when they block. Blocking can be caused by synchronization primitives (see below) or by native methods, which might not return right away. Luckily, all of the JVM's methods which might block will respond to calls to `Thread.interrupt()`, causing the formerly blocked thread to immediately throw an exception and canceling the operation that was previously underway. We used this mechanism with soft termination to signal blocking threads that we wished to kill.

However, in implementing soft termination, we found that some mechanism was still needed to determine which thread is blocking, and whether it was blocking on behalf of system code (which should not be interrupted) or on behalf of a codelet (which we *want* to interrupt). We chose to wrap blocking methods with code to register the current thread with the soft termination system as blocking before the call, and unregister it afterward. We use Java's stack inspection mechanism to determine the context of the blocking call (system versus codelet). The soft termination system could now interrupt blocking threads as necessary.

Synchronization The semantics of Java's `monitorenter` and `monitorexit` bytecode instructions and `synchronized` method modifier cannot be changed through bytecode rewriting. When a deadlock occurs in Sun's JDK, the only way to recover is by restarting the JVM. The JDK 1.4 debugging architecture provides a mechanism to address this (a debugger is allowed to forcibly pop activation records from a thread's stack), which might be useful to clean up after deadlock. This can similarly be used to terminate threads that are in a deadlock situation.

Another issue which soft termination had to deal with was the exact semantics of the `monitorenter` and `monitorexit` bytecodes, which acquire and release system locks, respectively. If these calls are not properly balanced, it becomes possible to lock a monitor in such a way that terminating the codelet will not cause the monitor to be released. Despite the fact that neither the JVM nor the Java language specifications allow such construction, current JVM bytecode verifiers accept such programs. Our soft termination system did not attempt to deal with this problem.

Verification We have seen several cases where our own code had to effectively extend the Java bytecode verifier in order to guarantee correctness of our system. We saw these issues with Java's synchronization and exception features. We also saw cases where Java's verifier got in the way of perfectly sound program transformations, particularly with regard to the restrictions on how Java's constructors invoke their super-class constructors.

Ideally, the Java bytecode verifier should be disentangled from the current class loading system to stand on its own. This would simplify the addition of new checks to the verifier, such as checks for undesirable exception and synchronization behavior, and it would make it easier to remove checks that, at least in our context, are unnecessary, such as the super-class constructor checks. Furthermore, a modular Java bytecode verifier would be quite useful to our systems as a mechanism for checking our input *before* we rewrite it, allowing us to make stronger assumptions about the quality of our input and reducing the opportunity for carefully crafted codelets to trick a code rewriting system into mistakenly outputting a rewritten codelet with more privilege than it should have been given.

3.3 "Global" Java Semantics

In many cases, particularly when we consider changes that effect the method signatures advertised by a class, we must consider how our changes interact with other Java classes, with native code, and even with classes that may not have been loaded into the system yet.

"Special" Methods and Classes Every JVM has certain methods and classes which are special in some way. Sun's JVM, for example, doesn't allow the rewriter to add fields to `java.lang.Class` or `java.lang.Object`. The NaturalBridge Java system is even more restrictive, as it is implemented, itself, in Java, and has many "special" Java classes that provide the necessary machinery to implement language internals. Calls to these (privileged) classes are quietly replaced with the (dangerous) primitive operations that they represent. Changes to these classes are simply ignored.

If a global transformation could be applied to *all* Java classes in a consistent way, such as the security-passing style transformation of SAFKASI, then the resulting system would be perfectly self-consistent. However, once special methods and classes are added, everything becomes more difficult. Now, the system must keep a list of special classes and methods and treat calls to them as special cases.

For SAFKASI, this meant that we could not pass our security context as an additional argument to all methods. Instead, we used thread-local storage to temporarily hold the security context, and then called the original, special method. If that method just returned, then everything would continue as normal. If that method were to make a call to another Java method, then it would be using the original Java method signatures, without the additional argument for passing security contexts.

To support this, we were forced to add wrapper methods for every method in the system. These wrappers existed solely to receive calls from special methods, fetch the security context from the thread-local storage, and then invoke the proper target

method. Luckily, this wrapper technique proved to be quite general. It also supports native method up-calls and Java reflection (although we never tried to hack Java reflection such that it would see all the classes as if they had never been rewritten). By pairing wrappers, one to convert from security-passing style methods to special methods, and one to convert from regular methods back to security-passing style methods on up-calls from special code, we were able to maintain the illusion that the whole system supported security-passing style calling conventions. We used the same strategy in our transactional rollback system to manage the transactional state, normally passed as an additional argument to all methods.

Inheritance Java's class inheritance is also a complicating factor in global transformations. When a subclass overrides a special method, as described above, it inherits the "specialness" of the method. For example, `java.lang.Object.hashCode()` is a native method. Any other class can override this, providing a Java (or native) implementation. At an arbitrary call site, where `java.lang.Object.hashCode()` is invoked, there may not be enough information to determine a more specific type for the callee. Thus, the caller must assume the worst case: the callee is special. This requires saving the security context, and then making a call to the special method. Of course, if the concrete type was something other than `java.lang.Object`, and it had, in fact, overridden `hashCode()`, then control would enter a wrapper method which would recover the security context and call back into the world of rewritten code.

Open World Java Java is an *open world* system. That is, classes can be loaded and the class hierarchy modified at run-time. The only restrictions that Java makes are that, for a class to be loaded, all of its super-classes must already be loaded. Even with this restriction, the open world assumption complicates code analysis. It is no surprise, then, that many systems that change Java's semantics, including SAFKASI, assume a closed world (that once the system starts running, no more classes will be loaded). SAFKASI performs a class hierarchy analysis [27], reading every class in the system into a large data structure that allows a number of queries. SAFKASI uses this to determine if all possible callees of a given call site are "normal" methods. For the `java.lang.Object.hashCode()` example above, SAFKASI would conservatively conclude that the callee might be special, and therefore the security context should be saved. SAFKASI's analysis is fairly simplistic, as it doesn't take advantage of data flow and control flow information to narrow its idea of the type of a given callee. Still, even this simple flow-invariant analysis provided for significant optimizations.

Because Java is actually an open world, every Java class to be inserted into the system can potentially invalidate a judgment made by the optimizer. As a result, the optimizer would need to *back out* the optimizations that now reside in code previously loaded into the system. As we discussed in Section 3.1, such functionality is only now becoming available in the JDK 1.4 debugging architecture, and might make such optimizations possible, even in an open world.

Bootstrapping Bootstrapping presents a unique problem to bytecode rewriting systems. When the JVM is launched, it normally proceeds through the initialization of

its core classes before loading any applications. These core classes are, by necessity, carefully designed to avoid circular dependencies in their static initializers. Circular dependencies can be particularly hazardous in JVMs implemented themselves in Java, where the very first classes to be initialized are all "special" to the system in some way, and are very fragile with respect to changes.

In the implementation of soft termination, we largely did not need to worry about bootstrapping because we only had to transform codelets, not the entire system. For transactional rollback and SAFKASI, we had to transform everything (or as much of everything that was not special, in some fashion or another). In practice, when implementing SAFKASI on the NaturalBridge Java system, we had a large list of "special" classes that we could not touch. Likewise, we had to make sure the bootstrapping process could complete before any of the SAFKASI-specific classes were initialized. Our solution was to pass `null` in our security context argument and to only allocate a security context, and thus cause the SAFKASI system to be initialized, right before starting a codelet. For transactional rollback, we have the benefit that we support two modes of operation: with and without transactions. As a result, the JVM can initialize itself normally, and we only transition to the transactional world when we are about to start a codelet. Transactional rollback, in practice, proved much simpler to debug.

4 Optimizing Bytecode Rewriting

In the development of our systems, we used extensive profiling to determine where optimization would be necessary and guess at what optimizations might be profitable. For example, if we suspect that, for a large number of call-sites, the callee has a desirable property for optimizing the call-site, then we would instrument the call-sites to count exactly how many callees happen to have the property in question, and at which call-sites. Using this style of analysis, we were able to rapidly focus our attention on the optimizations that might matter for our systems. Of course, running our programs with such profiling slowed them down, but these profiling checks are only actually included during code rewriting when we wish to gather profiling data.

Soft Termination. The unoptimized design of soft termination required a termination check to be inserted before every call site. For methods with multiple call sites, there would be one termination check added per call site. We measured, through profiling, that on average there was more than one termination check per method invocation. This led us to "push" the termination checks from the call site to the head of the callee, thus reducing the number of termination checks.

Next, we could easily determine which methods, having no backward branches or outward method invocations, are guaranteed to return in a finite time. For these methods, the termination check at the beginning of the method is unnecessary and can be omitted. In some cases, this as much as halved the overhead of soft termination relative to the overhead of the unoptimized soft termination system.

Transactional Rollback. In our transactional rollback system, we learned a number of seemingly obvious facts through profiling. We observed that, for most lock acquire

operations, the transaction acquiring the lock was the same transaction already holding the lock. We also observed that, by far, the most common object to be locked is the `this` object.

These measurements led us to some simple optimizations with profound effects on system performance. By checking if the lock to be acquired is already held by the current transaction, we generally saved at least one method call, sometimes more. Likewise, by checking if a method contains multiple locking operations on `this` and consolidating them to a single operation at the head of the method, we were able to remove a significant number of lock operations. These two optimizations alone bought us a 25% speed improvement.

SAFKASI. SAFKASI performed a number of different optimizations aimed at lowering the cost of computing transitions in the security context information for each method invocation. We measured that from 10 to 37% of method invocations were to leaf methods (with no outgoing method calls), leading us to optimizations similar to those described above for soft termination. Likewise, after implementing a global class hierarchy analysis, we were able to determine that we could statically predict the security context information for 10 to 76% of the remaining call-sites. Combining these with a simple one-reference cache of the last security context transition computed at a given call-site, we were able to optimize away virtually all security context transitions, leading us to focus our efforts on other aspects of the system.

We then studied the operations for storing and retrieving context information from thread-local storage. By noting the current method every time one of these operations was performed, and keeping one counter for every method in the system, we quickly realized that most of the context stores were focused on calls to a relatively small number of methods: generally these were methods that were called from all over the system, such as `Object.toString()` on the result of `Hashtable.get()`. Further optimization would require implementing control and data flow analysis in order to infer, for example, what the concrete type emerging from the Hashtable might be, rather than just `java.lang.Object`. Having access to the analysis that is, no doubt, performed within the JVM's internal optimizer may have enabled such optimization.

5 Conclusions

Bytecode rewriting is an extremely powerful mechanism for modifying the behavior of Java by modifying the Java classes directly. It allows for essentially arbitrary changes to Java's semantics. It is portable in that it allows a system based on bytecode rewriting to run on any JVM. There are a number of limitations to bytecode rewriting, as well. These stem from the fact that we cannot directly affect how the JVM processes a given bytecode instruction or set of bytecode instructions, except using a relatively small number of restricted interfaces.

Considering these limitations, bytecode rewriting systems will never be able to match the capabilities and performance of customized JVMs, which have access "under the hood" of the JVM. However, customized run-time systems suffer from a lack of portability. In addition, bytecode rewriting systems benefit from optimization courtesy

of the JVM. With appropriate specialized optimizations of the sort discussed above, a system based on bytecode rewriting could be competitive performance- and capability-wise with a system based on language run-time system customization.

Acknowledgement. We would like to thank the anonymous JVM 2002 and ISSS 2002 reviewers for their helpful suggestions. This work is supported by NSF Grant CCR-9985332 and a Texas ATP grant.

References

1. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading, Massachusetts (1996)
2. Wallach, D.S., Felten, E.W., Appel, A.W.: The security architecture formerly known as stack inspection: A security mechanism for language-based systems. ACM Transactions on Software Engineering and Methodology **9** (2000) 341–378
3. Rudys, A., Wallach, D.S.: Termination in language-based systems. ACM Transactions on Information and System Security **5** (2002) 138–168
4. Rudys, A., Wallach, D.S.: Transactional rollback for language-based systems. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, Washington, DC (2002)
5. Pandey, R., Hashii, B.: Providing fine-grained access control for Java programs. In Guerraoui, R., ed.: 13th Conference on Object-Oriented Programming (ECOOP'99). Number 1628 in Lecture Notes in Computer Science, Lisbon, Portugal, Springer-Verlag (1999)
6. Erlingsson, U., Schneider, F.B.: IRM enforcement of Java stack inspection. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, Berkeley, California (2000) 246–255
7. Chander, A., Mitchell, J.C., Shin, I.: Mobile code security by Java bytecode instrumentation. In: 2001 DARPA Information Survivability Conference & Exposition (DISCEX II), Anaheim, CA, USA (2001)
8. Hawblitzel, C., Chang, C.C., Czajkowski, G., Hu, D., von Eicken, T.: Implementing multiple protection domains in Java. In: USENIX Annual Technical Conference, New Orleans, Louisiana, USENIX (1998)
9. Binder, W.: Design and implementation of the J-SEAL2 mobile agent kernel. In: 2001 Symposium on Applications and the Internet, San Diego, CA, USA (2001)
10. Czajkowski, G., von Eicken, T.: JRes: A resource accounting interface for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia (1998) 21–35
11. Clausen, L.R.: A Java bytecode optimizer using side-effect analysis. Concurrency: Practice and Experience **9** (1997) 1031–1045
12. Nystrom, N.J.: Bytecode level analysis and optimization of Java classes. Master's thesis, Purdue University (1998)
13. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13** (1991) 451–490
14. Lee, H.B., Zorn, B.G.: BIT: A tool for instrumenting java bytecodes. In: USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA (1997)
15. Cohen, G., Chase, J., Kaminsky, D.: Automatic program transformation with JOIE. In: Proceedings of the 1998 Usenix Annual Technical Symposium, New Orleans, Louisiana (1998) 167–178

16. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot – a Java bytecode optimization framework. In: Proceedings of CASCON 1999, Mississauga, Ontario, Canada (1999) 125–135
17. Sakamoto, T., Sekiguchi, T., Yonezawa, A.: Bytecode transformation for portable thread migration in Java. In: Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA). (2000) 16–28
18. Marquez, A., Zigman, J.N., Blackburn, S.M.: A fast portable orthogonally persistent Java. Software: Practice and Experience Special Issue: Persistent Object Systems **30** (2000) 449–479
19. Welch, I., Stroud, R.: Kava – a reflective Java based on bytecode rewriting. In: Lecture Notes in Computer Science 1826. Springer-Verlag (2000)
20. Deutsch, P., Grant, C.A.: A flexible measurement tool for software systems. In: Information Processing 71: Proceedings of the IFIP Congress. Volume 1., Ljubljana, Yugoslavia (1971)
21. Gong, L.: Inside Java 2 Platform Security: Architecture, API Design, and Implementation. Addison-Wesley, Reading, Massachusetts (1999)
22. NaturalBridge, LLC: BulletTrain Java Compiler. (1998) `http://www.natural-bridge.com`.
23. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley, Reading, Massachusetts (1996)
24. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño virtual machine. IBM System Journal **39** (2000)
25. Price, D., Rudys, A., Wallach, D.S.: Garbage collector memory accounting in language-based systems. Technical Report TR02-407, Department of Computer Science, Rice University, Houston, TX (2002)
26. Blackburn, S.M., Singhai, S., Hertz, M., McKinley, K.S., Moss, J.E.B.: Pretenuring for Java. In: OOPSLA 2001: Conference on Object-Oriented Programming Systems, Languages, and Applications. Volume 36 of ACM SIGPLAN Notices., Tampa Bay, Florida (2001) 342–352
27. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95), Århus, Denmark (1995)