A Type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language

Pratibha Permandla  Chandrasekhar Boyapati

Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109
{pratibha,bchandra}@eecs.umich.edu

Abstract
In previous work on SafeJava we presented a type system extension to the Java source language that statically prevents data races and deadlocks in multithreaded programs. SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. SafeJava thus offers a promising approach for making multithreaded programs more reliable. This paper presents a corresponding type system extension for the Java virtual machine language (JVML). We call the resulting language SafeJVML. Well-typed SafeJVML programs are guaranteed to be free of data races and deadlocks. Designing a corresponding type system for JVML is important because most Java code is shipped in the JVML format. Designing a corresponding type system for JVML is nontrivial because of important differences between Java and JVML. In particular, the absence of block structure in JVML programs and the fact that they do not use named local variables in the way Java programs do make the type systems for Java and JVML significantly different. For example, verifying absence of races and deadlocks in JVML programs requires performing an alias analysis, something that was not necessary for verifying absence of races and deadlocks in Java programs. This paper presents static and dynamic semantics for SafeJVML. It also includes a proof that the SafeJVML type system is sound and that it prevents data races and deadlocks. To the best of our knowledge, this is the first type system for JVML that statically ensures absence of synchronization errors.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs
D.2.4 [Software Engineering]: Program Verification

General Terms
Languages, Verification

Keywords
SafeJava, Data Races, Deadlocks, Ownership Types

1. Introduction
Multithreaded programming is becoming a mainstream programming practice. But multithreaded programming is difficult and error prone. Multithreaded programs synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to data races or deadlocks. A data race occurs when two thread concurrently access the same shared data, at least one of the accesses is a write, and there is no synchronization to separate the accesses. A deadlock occurs when there is a set of threads such that every thread in the set is waiting on a lock held by another thread in the set. Synchronization errors in multithreaded programs are timing-dependent, nondeterministic bugs, and are among the most difficult programming errors to detect, reproduce, and eliminate.

In previous work on SafeJava [5, 6, 9] we presented a static type system for multithreaded programs. Well-typed SafeJava programs are guaranteed to be free of data races and deadlocks. The basic idea is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. SafeJava allows programmers to specify this locking discipline in their programs in the form of type declarations. The type checker then statically verifies that a program is consistent with its type declarations. The SafeJava type system also enforces object encapsulation [5, 7], which is key to enable local reasoning in object-oriented programs. For example, a Stack object a that is implemented using a linked list. Local reasoning about the correctness of the Stack implementation is possible if objects outside a do not directly access the list nodes, i.e., the list nodes are encapsulated within a. SafeJava uses a variant of ownership types [13, 12, 3] to declare that a owns all the list nodes. The type system then statically ensures that the list nodes are encapsulated within a. Object encapsulation is useful for safe multithreading because the lock that protects an object can also protect the objects encapsulated within that object.

Our experience suggests that SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. In addition, the type declarations in SafeJava programs serve as documentation that lives with the code and is checked throughout the evolution of code. The SafeJava type system thus provides significant software engineering benefits and offers a promising approach for improving the reliability of multithreaded and object-oriented programs.

This paper presents a corresponding type system for (a subset of) the Java virtual machine language (JVML). We call the resulting language SafeJVML. Well-typed SafeJVML programs are guaranteed to be free of data races and deadlocks. Well-typed programs are also guaranteed to enforce object encapsulation. This paper presents the static and dynamic semantics of SafeJVML, and includes a proof that the SafeJVML type system is sound and that it prevents data races and deadlocks and enforces encapsulation.

Designing a corresponding type system for JVML is important because it is the format of choice for shipping code. Systems that download untrusted JVML programs first perform bytecode verification to ensure absence of memory errors before running the programs. With our proposed extension to the JVML type sys-
thread one must use alias analysis to statically ensure that

Link to static analysis.

getfield monitorenter putfield

one must use alias analysis to statically ensure that

getfield JVML code in Figure 2. To check that the JVML code is race-free,

from Account

jects happen within the block of code where the locks on the

balance

to

transfer

then statically verify that the

balance

to

transfer

met, a bytecode verifier can also statically ensure the absence of
data races and deadlocks in a program before running it. Moreover,
many code bases have a combination of Java source code and
JVML code. Verifying the absence of races and deadlocks in such a
code base requires corresponding race-free and deadlock-free type
systems for both Java and JVML.

Designing a corresponding type system and a syntax-directed type
checker for JVML is nontrivial because of important differences be-
tween Java and JVML. In particular, the absence of block structure in
JVML programs and that they do not use named local variables like
Java programs make the type systems for Java and JVML signi-
ficantly different. For example, verifying absence of races and
deadlocks in JVML programs involves performing an alias analy-
sis, something we did not have to do for verifying Java programs.

Consider the transfer method in Figure 1. Suppose there are
type annotations (not shown in the figure) that declare that every
Account object is protected by its own lock. A type checker can
then statically verify that the transfer method is race-free because
the accesses to the balance field of the to and from Account ob-
jects happen within the block of code where the locks on the to and
from Account objects are held. Now consider the corresponding
JVML code in Figure 2. To check that the JVML code is race-free,
one must use alias analysis to statically ensure that the getfield
and putfield instructions operate on the same objects on which
the locks are obtained using monitorenter. Moreover, one must
statically ensure that getfield and putfield instructions happen af-
after the corresponding monitorenter instructions and before the
corresponding monitorexit instructions, something that is non-
trivial in general if the code is not block structured and uses gotos.

To the best of our knowledge, this is the first type system for
JVML that statically prevents data races, deadlocks and encap-
sulation violations. This paper combines ideas from four different systems—
i) formalization of the JVML type system [4, 20, 29], ii) type systems
for preventing data races and deadlocks in Java programs [5, 9, 6,
16] iii) ownership types for enforcing object encapsulation [1, 3,
7, 13], and iv) type systems for JVML for statically ensuring that

Figure 1. A transfer method in Java

static void transfer(Account, Account, int); 1
static void transfer(Account from, Account to, int x) { 2

1: load 0 17: putfield #2; //balance:I 3: load 3 21: load 0

2: store 3 20: load 0 4: monitorenter 22: getfield #2; //balance:I

3: load 3 27: putfield #2; //balance:I 5: load 1 25: load 2

4: monitorenter 22: getfield #2; //balance:I 6: store 4 26: sub

5: load 1 25: load 2 7: load 4 27: putfield #2; //balance:I

6: store 4 26: sub 8: monitorenter 30: load 4

7: load 4 27: putfield #2; //balance:I 9: monitorenter 32: monitorexit

8: monitorenter 30: load 4 10: load 1 32: monitorexit

9: monitorenter 30: load 4 11: load 1 44: load 3

10: load 1 32: monitorexit 12: getfield #2; //balance:I 45: monitorexit

11: load 1 44: load 3 15: load 2 46: return

12: getfield #2; //balance:I 45: monitorexit 16: add

Figure 2. The transfer method in Figure 1 in JVML (excluding ex-
pception handling)

tem, a bytecode verifier can also statically ensure the absence of
data races and deadlocks in a program before running it. Moreover,
many code bases have a combination of Java source code and
JVML code. Verifying the absence of races and deadlocks in such a
code base requires corresponding race-free and deadlock-free type
systems for both Java and JVML.

Designing a corresponding type system and a syntax-directed type
checker for JVML is nontrivial because of important differences be-
tween Java and JVML. In particular, the absence of block structure in
JVML programs and that they do not use named local variables like
Java programs make the type systems for Java and JVML signi-
ficantly different. For example, verifying absence of races and
deadlocks in JVML programs involves performing an alias analy-
sis, something we did not have to do for verifying Java programs.

Consider the transfer method in Figure 1. Suppose there are
type annotations (not shown in the figure) that declare that every
Account object is protected by its own lock. A type checker can
then statically verify that the transfer method is race-free because
the accesses to the balance field of the to and from Account ob-
jects happen within the block of code where the locks on the to and
from Account objects are held. Now consider the corresponding
JVML code in Figure 2. To check that the JVML code is race-free,
one must use alias analysis to statically ensure that the getfield
and putfield instructions operate on the same objects on which
the locks are obtained using monitorenter. Moreover, one must
statically ensure that getfield and putfield instructions happen af-
after the corresponding monitorenter instructions and before the
corresponding monitorexit instructions, something that is non-
trivial in general if the code is not block structured and uses gotos.

To the best of our knowledge, this is the first type system for
JVML that statically prevents data races, deadlocks and encap-
sulation violations. This paper combines ideas from four different systems—
i) formalization of the JVML type system [4, 20, 29], ii) type systems
for preventing data races and deadlocks in Java programs [5, 9, 6,
16] iii) ownership types for enforcing object encapsulation [1, 3,
7, 13], and iv) type systems for JVML for statically ensuring that

each thread and cannot be accessed by any other thread. All other objects
are potentially shared between multiple threads. Figure 3 presents

Figure 3. An ownership relation

E1. The owner of an object does not change over time.
E2. The ownership relation forms a tree rooted at world.
E3. If object z owns y but z ̸⪰ x, then x cannot access y.
R4. To safely access an object, a thread must hold the lock on the
root owner of that object. (r is the root owner of an object o
iff r ⪰ o and world directly owns r.)
R5. Every thread implicitly holds the lock on its corresponding
thisThread owner. A thread can access objects owned by
its thisThread without synchronization.
D6. Every lock belongs to some lock level. The lock level of
a lock does not change over time. The lock levels form a
partial order.
D7. To acquire a new lock of lock level l, the levels of all the
locks held by the thread must be greater than l.

Figure 4. SafeJava properties

methods release all locks they acquire and no other lock [26, 23]—
and achieves what neither of these systems individually achieves.

We also note in passing that any type system that guarantees race
freedom also eliminates all the complex issues associated with the
use of weak memory consistency models [28]. A detailed expla-
nation of this issue can be found in [2]. The rest of the paper is
organized as follows. Section 2 presents an overview of SafeJava.
Section 3 presents SafeJVML, including its dynamic and static se-
manitics and a soundness proof. Section 4 describes related work.

2. Overview of SafeJava

This section presents an overview of a core subset of SafeJava for
preventing data races [9] and deadlocks [6] and encapsulation vi-
lations [7]. The key to the type system is the concept of object
ownership. Every object has an owner. An owner can be owned by
another object, by a special per-thread owner called thisThread,
or a global owner called world. We use the notation o1 ⪰ o2 to de-
note that o1 directly or transitively owns o2 or o1 is the same as o2.
The relation ⪰ is thus the reflexive transitive closure of the owns
relation. If thisThread ⪰ o, then o is local to the corresponding
thread and cannot be accessed by any other thread. All other objects
are potentially shared between multiple threads. Figure 3 presents
an example ownership relation. We draw an arrow from object x to
object y if x owns y. Our type system statically verifies that a pro-
gram respects the properties shown in Figure 4. Properties E1-E3
ensure encapsulation. Properties R4 and R5 prevent races.

Figure 5 shows a TStack program in SafeJava. A TStack is a
stack of T objects. A TStack is implemented using a linked list.
A class definition in SafeJava is parameterized by a list of owners.
This parameterization helps programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. In Figure 5, the TStack class is parameterized by thisOwner and TOwner. thisOwner owns the TStack object and TOwner owns the T objects contained in the TStack. In general, the first formal parameter of a class always owns the object. In case of s1, the owner thisThread is used for both the parameters to instantiate the TStack class. It means that TStack s1 as well as all the T objects contained in the TStack are local to the main thread. In case of s2, the TStack is local to the main thread but the T objects contained in the TStack are potentially shared between multiple threads. In case of s3, both the TStack and the T objects contained in the TStack are potentially shared between multiple threads. The ownership relation for the TStack objects s1, s2, and s3 is depicted in Figure 6 (assuming the stacks contain two elements each). In SafeJava, a method can contain a requires clause that specifies the objects the method accesses that must be protected by externally acquired locks. Callers are required to hold the locks on the root owners (see Figure 4) of the objects specified in the requires clause before they invoke a method to avoid data races. The pop method assumes that the callers hold the lock on the root owner of the TStack object.

To prevent deadlocks, programmers partition all the locks in our system into a fixed number of lock levels and specify a partial order among the lock levels. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. A lock level is like a static field in Java; a lock level is a per-class entity rather than a per-object entity. But unlike static fields in Java, lock levels are used only for compile-time type checking and are not preserved at runtime. Programmers can specify a partial order among the lock levels using the < and > syntax in the lock level declarations. Since a program has a fixed number of lock levels, our type checker can statically verify that the lock levels do indeed form a partial order. Properties D6 and D7 in Figure 4 prevent deadlocks.

Note that the complete SafeJava language is more expressive than the core subset presented here, and supports most of the commonly used synchronization patterns. It also supports safe region-based memory management [10] and safe software upgrades [8]. A detailed description can be found in [5] and [6, 7, 8, 9, 10].

3. SafeJVML

This section presents SafeJVML, an extension to the Java virtual machine language (JVML) for statically preventing data races and deadlocks as well as for statically enforcing object encapsulation. To simplify the presentation of key ideas behind our approach, we describe our system formally in the context of a core subset of JVML. In particular, we avoid subroutines and object initialization because they are orthogonal to preventing synchronization and encapsulation errors. However, they can be easily added to our system using previous work on formalization of subroutines [11, 24, 30] and object initialization [19] for JVML.

Figure 7 shows the grammar for SafeJVML. The grammar is similar to the grammar for SafeJava [5] with respect to the class and method signatures. But the SafeJVML instructions are different from those of SafeJava. In particular, JVML programs are not block structured and do not use named local variables. This makes it difficult to design a syntax-directed type checker for JVML that tracks the relation between locks acquired and the objects they protect. To address this problem we use indexed types [26], which statically guarantee that all variables with the same indexed type ci are aliases. Indexed types [26] were previously used to statically ensure that the monitorenter and monitorexit instructions are matched along every program path. In this paper, we adopt the idea to ensure absence of races and deadlocks in JVML programs.

The SafeJVML instruction set closely resembles the JVM instruction set. The only difference is the format of methoddef shown in Figure 7. In SafeJVML, methodref also includes a requires clause which specifies the objects the method accesses that must be protected by externally acquired locks. Each xi in requires(x0, k) denotes the kth argument passed to the method. For example, requires(x0, x1) specifies that the this object (x0) and the first argument (x1) must be protected by externally acquired locks.

3.1 Dynamic Semantics

This section presents a small step operational semantics for SafeJVML. This is necessary to formally define the semantics of SafeJVML programs, and well as to state and prove the type soundness.
the runtime owners
fields in an object of type cn
records. The definitions are shown in Figure 8. The heap is modeled as a partial function $h$ the static formal owner parameters with this object. We use the notation $T$ the special field $T$ the locks specified in the transition however applies to every configuration that contains the only include the components that participate in the transition. The Figure 10 presents the dynamic semantics for SafeJVML and Figure 9. Auxiliary definitions for dynamic semantics

we assume there is a separate intraprocedural type inference phase that infers the types of local variables at every program point. This paper only describes the type checking rules. Type inference can be performed by solving the constraints generated by the type checking rules. The advantage of separating type inference from type checking is that it reduces the size of the trusted computing base; a bug in type inference cannot compromise a JVM, only a bug in type checking can. Moreover, type checking becomes syntax directed. We also assume the SafeJava to SafeJVML compiler generates programs according to the grammar in Figure 7. That is, the compiler preserves the type annotations on class and method signatures.

The core of our type system is a set of rules for reasoning about the typing judgment: $P, E, F, S, LS, L \vdash M$ $P$ denotes the program that is being checked. It contains the information about class definitions. The typing environment $E$ tracks the owners and constraints which are in scope. The typing environment contains the declared owner parameters, the declared constraints among owners, and the declared $Locks$ clause in scope:

$E := \emptyset \mid E, fowner F \mid E, constr \mid E, locks(cn, ls)$

$F, S, LS,$ and $L_{min}$ provide respectively the types of local variables, the types of stack slots, the locks that are statically known to be held, and the sequence of minimum lock levels at every program point. That is, $F_i$ is the map from local variables to types at $i^{th}$ instruction. $S_i$ is a sequence of types of the operand stack at $i^{th}$ instruction. $LS_i$ is a multi-set of indexed object types denoting the locks held at instruction $i$. $L_{min}$ is a sequence of $l_{min}$’s. Recall lock levels from Section 2. The definition of $l_{min}$ is as follows:

$l_{min} := \infty \mid cn.l \mid lub(cn_1, l_1 ... cn_k, l_k)$

By definition, $lub(cn_1, l_1 ... cn_k, l_k) \geq cn.l$ $\forall i \geq 1, k$. $LUB(\ldots)$ is not computed—it is an expression used as such for type checking. The lock level $\infty$ denotes that the thread currently holds no locks.

Figures 11 presents the static semantics for the instructions in SafeJVML. The full set of typing rules are in the appendix. The judgment $P, E, F, S, LS, L_{min}, i \vdash M$ denotes that instruction $i$ satisfies all type constraints. $M[i]$ denotes the $i^{th}$ instruction of the method with methodref $M$. We use the notation that for any type $t$, $t.owners = o_{ts}$, if $t = cn(o_{ts})$. Also, $[o_1/f_1][o_2/f_2]...[o_m/f_m]$ denotes the type $t$ in which the formal owner parameters are replaced with actual owner parameters.

Figure 12 illustrates the types at every program point for the $transfer$ method shown in Figure 2. We use this example to explain few of our typing rules. Like we mentioned before, we use indexed types to keep track of aliases. The indexed type $c_i$ is the type of

$$
\begin{align*}
C & := \Phi; h \\
\Phi & := T \Phi \mid \epsilon \\
T & := (A) \\
A & := \{M, p, f, s, ls\} \cup (A \mid \epsilon) \\
h & : location \rightarrow (fd = v_1, g_1 = w_1, level = cn_{\epsilon}, l_{\epsilon}) \cup (1 \leq n) \\
\text{Figure 8. SafeJVML execution state}
\end{align*}
$$

$$
\begin{align*}
type(h, v) & = \begin{cases}
\text{int, if } v \text{ is an integer} \\
\text{t, if } v \in \text{location and } h[v] = (\_\_\_\_\_\_) \\
\end{cases} \\
lock(h, v) & = \begin{cases}
\text{thisThread, if } (type(h, v) = cn(\text{thisThread})) \\
\text{v, if } (type(h, v) = cn(\text{world-CN})) \\
\text{lock(h, v'), if } (type(h, v) = cn(v', \ldots)) \\
\end{cases} \\
level(h, g) & = \begin{cases}
\text{cn.l, if } g = \text{world-CN.l} \\
\infty, \text{otherwise} \\
\end{cases} \\
RO(o, g) & = \begin{cases}
\text{thisThread, if } g = \text{thisThread} \\
\text{world, if } g = \text{world-CN.l} \\
o, \text{if } g = \text{this} \text{[h|g], otherwise} \\
\end{cases} \\
f_0 & : \text{function mapping local variables to arbitrary values}
\end{align*}
$$

The new instruction creates a new object and initializes its fields to default values. It also initializes fields $g_{1..n}$ with runtime owners of the object. To access the runtime owners, it uses a map $RO$ shown in Figure 9, which takes an object and a static owner parameter and returns the corresponding runtime owner. The $start$ instruction starts a new thread with the lock set that contains only thisThread, because the new thread does not inherit any locks from its parent thread. The control of the new thread is transferred to its $start$ method. Figure 10 presents these and other rules formally.

### 3.2 Static Semantics

This section describes the static semantics of SafeJVML. Following standard practice in JVM type system formalizations [20, 26, 30], we have the following congruence rule. Below, $\Phi_1, \Phi_2, h' \rightarrow (\text{Locks}[\Phi_1] \cup \text{Locks}[\Phi_2]) \cap \text{Locks}[\Phi] = \emptyset$.

$$
\begin{align*}
\Phi_1; h & \rightarrow \Phi_2; h' \\
(\text{Locks}[\Phi_1] \cup \text{Locks}[\Phi_2]) \cap \text{Locks}[\Phi] & = \emptyset \\
\Phi_1 \cup \Phi_2; h & \rightarrow \Phi_2 \cup \Phi_1; h' \\
\end{align*}
$$

The new instruction creates a new object and initializes its fields to default values. It also initializes fields $g_{1..n}$ with runtime owners of the object. To access the runtime owners, it uses a map $RO$ shown in Figure 9, which takes an object and a static owner parameter and returns the corresponding runtime owner. The $start$ instruction starts a new thread with the lock set that contains only thisThread, because the new thread does not inherit any locks from its parent thread. The control of the new thread is transferred to its $start$ method. Figure 10 presents these and other rules formally.
variables whose value of type \( c \) was first copied at \( i \)th instruction. In a well-typed program, all variables have the same type \( c_i \) are guaranteed to be aliases. Consider the rule for \texttt{load} in Figure 11. If it is the first copy of the value, then the type is changed; the type of the object is tagged with the program point at which the \texttt{load} is performed. For example, in Figure 12, one of the \texttt{Acccount} object's type is tagged with PC 1, the instruction at which the object is first copied on to the stack. The other \texttt{Acccount} object's type is tagged with PC 5. Successive copies preserve the type of the first copy.

We define an indexing operation over types \( t \) to mark types when variables are copied. Let \( t_i \), where \( i \) is an integer, be the following:

\[
t_i = \begin{cases} 
    t, & \text{if } i=0 \\
    c_i, & \text{if } i=1 \text{ (first copy of the variable changes the type)} \\
    c, & \text{if } i>1 \text{ (successive copies keep the type of the first copy)}
\end{cases}
\]

We also define \texttt{Index} and \texttt{Type} as partial functions from types to integers and types respectively. The notation \( t \) indexed types.

\[
\text{Index}[c_i] = i, \quad \text{and} \quad \text{Type}[c_i] = c_i, \quad \text{if } c = \text{undefined otherwise}
\]

Before we proceed, we explain the auxiliary function \texttt{Lock}(\( \hat{t} \)) (formally defined in the appendix). A lock is an object directly owned by world. \texttt{Lock}(\( \hat{t} \)) denotes the lock that protects an object with indexed type \( \hat{t} \). If the owner of an object is a formal owner parameter, then we cannot determine the root owner of the object from within the static scope of the enclosing class. In that case, we define the root owner of the object with indexed type \( \hat{t} \) to be \( \texttt{L}(\hat{t}) \). Note that \( \texttt{L}(\hat{t}) \) is not computed—it is used as such for type checking.

The rule for acquiring a new lock using \texttt{monitorenter} in Figure 11 checks that the top of the stack is a lock of some lock level \( \texttt{cn} \cdot \hat{t} \) that is less than \( \texttt{m} \texttt{in} \). The rule also ensures that after the instruction, \( \texttt{cn} \cdot \hat{t} \) is stored on the top of \( \texttt{L}_{\texttt{min}+1} \) sequence. The rule for \texttt{getfield} in Figure 11 checks that the class declares or inherits the field and that the type on the top of the stack matches the type of the class in which the field is declared. It also checks that the thread holds the lock on the \texttt{root owner} (see Figure 4) of the object.

Going back to our example in Figure 12, the thread acquires the lock on \texttt{Account1} and \texttt{Account2} objects before accessing their balance fields. By consistently acquiring the lock on an object before accessing its fields, the potential for data races is avoided. There are two points to note in this example. One is that the type system statically tracks that the \texttt{monitorenter} operations are performed on the same objects whose fields are accessed by the \texttt{getfield} instructions. The second point is to note that the type system statically tracks that each \texttt{getfield} operation is performed after the corresponding \texttt{monitorenter} operation and before the corresponding \texttt{monitorexit} operation on the same object.

The rule for invoking a method using \texttt{invokevirtual} in Figure 11 checks that the arguments are of right type and that the thread holds the lock on the root owner (see Figure 4) of the object.

3.3 Soundness

This section provides a proof that the SafeJVML type system is sound and that well-typed SafeJVML programs do not have data races or deadlocks or encapsulation errors. We first define a good machine state configuration. We use the notation \( P \vdash h \hat{t} \) to denote that the heap \( h \) is well-typed. The rules for mapping run-
Figure 11. Static semantics for SafeJVML instructions
time values in the heap to types are given at the end of appendix. GoodConfiguration$(P, \Phi, h)$ states that given program $P$ and heap $h$, the thread set $\Phi$ is well-typed iff for every activation record $(M, pc, f, s, ls) \in \Phi$ associated with instruction $f$, the conditions in Figure 13 hold with respect to the static type information $F$, $S$, and $LS$ for the method $M$, and where $E$ provides the environment of the call site, and $\Phi$ describes the locking constraints.

We now formally state and prove the theorems.

\textbf{Theorem 1 (SafeJVM Preservation)}

Suppose $P \vdash wt$. Then, $\forall \Phi, \Phi', h, h', P \vdash h wt$ and GoodConfiguration$(P, \Phi, h)$ and $P \vdash \Phi; h \rightarrow \Phi'; h'$, then $P \vdash h' wt$ and GoodConfiguration$(P, \Phi', h')$.

\textbf{Proof:} We sketch the proof for getfield $\| \text{cn}(f_1 \ldots n), f, d, t \|_F$ instruction to motivate the structure of the invariants. First, we show that the execution of getfield $\| \text{cn}(f_1 \ldots n), f, d, t \|_F$ instruction in a machine configuration $\Phi; h$, where $P \vdash h wt$ and GoodConfiguration$(P, \Phi, h)$, yields a new well-typed heap. For getfield, this is trivial to show since $h$ is not modified. In general, the heap updates respect three properties: the types of records never change, values written into heap records have the same types as the overwritten values, and any new records introduced by allocation are well-typed records. If an instruction changes a heap $h$ to $h'$ according to these rules, then $h'$ will be well-typed.

Next we show that the execution of getfield instruction preserves all the GoodConfiguration invariants listed in Figure 13. Suppose a getfield instruction moves the virtual machine from $(M, pc, f, o.s, s, ls, A); h$ to $(M, pc + f, v, s, ls, A); h$ where $v = h[a]$. $\| \text{cn}(f_1 \ldots n), f, d, t \|_F$. Further suppose that $E, F, S, and LS$ comprise the type information used to show that $P, E, F, S, LS, L_{\min}, pc \vdash M$. We proceed by showing that all the conditions in Figure 13 hold. All conditions except C4 hold trivially since the getfield instruction does not affect invariants. In fact, the only instruction that affects Condition C1 is the new instruction. It is easy to show that new preserves this invariant—every object has a unique owner and the ownership relation forms a tree before the execution of new, therefore adding a child to one of the nodes of the tree during new’s execution preserves the tree structure.

Coming back to the getfield instruction, the only condition that is affected by it is Condition 4 which states that $P, h \vdash s : \text{RunTimeType}(s, \text{Sp}_{pc+1})$. From the static type checking rule for getfield instruction, we have $P, E \vdash t.S_{pc} \vdash \text{cn}(o_1 \ldots n), \text{Sp}_{pc+1}$ for some $t$ and $\text{cn}(o_1 \ldots n)$. Since $P, h \vdash o.s' : \text{RunTimeType}(o.s', \text{Sp}_{pc})$, $P, h \vdash o : \text{RunTimeType}(o, \text{cn}(o_1 \ldots n))$, and $P, h \vdash v : \text{RunTimeType}(v, t)$, we can conclude that $P, h \vdash v.s' : \text{RunTimeType}(v.s', \text{Sp}_{pc+1})$. Thus, $P, h \vdash s : \text{RunTimeType}(s, \text{Sp}_{pc+1})$. Therefore the execution of getfield preserves all the invariants in Figure 13.

\textbf{Theorem 2 (SafeJVM Progress)}

Let $P \vdash wt$ and $\forall \Phi, h, P \vdash h wt$ and GoodConfiguration$(P, \Phi, h)$ hold, then either:

i) $\exists \Phi', h', P \vdash \Phi; h \rightarrow \Phi'; h'$ (progress), or

ii) $(\forall T \in \Phi)(T = \langle A \rangle \land A = e)$ (normal termination), or

iii) $\exists T \in \Phi, s.t. T$'s next instruction is a null pointer dereference.

\textbf{Proof:} We prove this by showing that if $P \vdash h wt$ and GoodConfiguration$(P, \Phi, h)$ hold, then either the program is stuck a deadlock state, or at least one thread is stuck attempting to dereference a null pointer, or at least one thread can make progress, or the activation record stack for every thread is empty. We later prove in Theorem 5 that a deadlock state is not possible because well-typed programs in SafeJVM are free of deadlocks. Thus, the above theorem holds. The details of the proof are similar to the details of the proof of Theorem 1 presented above.

\textbf{Theorem 3 (SafeJVM Encapsulation)}

An object $x$ can access an object owned by $o$ only if $(o \geq x)$. Theorem 1. SafeJVM Preservation Supp
C1. The ownership relation in the program forms a tree.
Recall that the owner of an object o is stored in \( h[o], g_0 \), where 
\( g_0 \) denotes the first formal parameter of o’s class.
C2. The owners of every object satisfy the owner constraints 
specified in object’s class.
That is, the runtime owners \( h[o], g_1, ..., h[o], g_n \) of an object o 
satisfy the constraints declared in its class definition.
Note that \( cn(h[o], g_1, ..., h[o], g_n) \) gives the runtime type of 
an object o whose static type is \( cn(g_1, n) \). We use RunTime-
Type\((v, t)\) to denote the runtime type of a value v whose static 
type is t. RunTimeType\((v, t) = t \) if t is an integer. RunTimeType
\((v, t) = cn(h[v], g_1, ..., h[v], g_n) \) if t = cn\((g_1, n)\).
C3. \( pc \in \text{Dom}(M) \)
C4. The stacks have values of the expected types.
That is, \( s=s_1\ldots s_k \) implies \( S_{pc} = t_1\ldots t_k \), and \( P, h \vdash v_i : \text{Run-
TimeType}(v_i, t_i) \). In short, \( P, h \vdash s : \text{RunTimeType}(s, S_{pc}) \).
C5. The local variables contain values of the expected types.
That is, \( \forall y \in \text{Dom}(F_{pc}) \) \( (F_{pc}[y] = t) => P, h \vdash f[y] : \text{Run-
TimeType(f[y], t)} \).
C6. The static and dynamic lock sets are consistent.
(a) \( \{o_j\} \subseteq \mathcal{L}_0 \) implies \( P, h \vdash o_j : \text{RunTimeType}(o_j, \hat{t}) \) 
and \( \{t\} \subseteq \mathcal{L}_0 \).
(b) \( \{o_j\} \subseteq \mathcal{L}_S \) implies \( P, h \vdash o_j : \text{RunTimeType}(o_j, \hat{t}) \) and
\( \{k - k'\}, \{\hat{t}\} \subseteq \mathcal{L}_S \), where \( k = h[o_j], l \) at program 
point \( pc \) and \( k' = h[o_j], l \) at program point 0.
\( \mathcal{L}_0 \) denotes the locks that are specified in the requires clause.
C7. Two variables with the same indexed type must be aliases.
Let,
(a) \( o = f(x) \), when \( F_{pc}[x] = \hat{t} \) or
\( o = v_j \), when \( s = v_1\ldots v_k, S_{pc} = t_1\ldots t_k \) and \( t_j = \hat{t} \)
(b) \( o' = f(y) \), when \( F_{pc}[y] = \hat{t} \) or
\( o' = v_j' \), when \( s = v_1\ldots v_k, S_{pc} = t_1\ldots t_k \) and \( t_j' = \hat{t} \)
If \( \hat{t} = \hat{t}' \), then \( o = o' \). Furthermore, if \( \hat{t} \in \mathcal{L}_S \), then \( o \in \mathcal{L}_S \).
C8. The static and dynamic lock levels of the locks are consistent.
If \( P, h \vdash o : cn(o_1 : l, o_2..n) \), then \( h[o].level = l \)

Figure 13. Properties of a good machine state configuration

Theorem 4 (SafeJVML DataRaceFreedom)
Well-typed programs in SafeJVML are free of data races.
Proof: The type checking rules for SafeJVML ensure that every thread 
holds the lock protecting an object in its static lock set \( \mathcal{L}_S \) 
before accessing the object. The \textit{GoodConfiguration} judgment en-
sures consistency between dynamic and static entities. Together, 
they ensure that every thread holds the lock protecting an object
in its dynamic lock set \( \mathcal{L}_S \) before accessing the object. Well-typed 
SafeJVML programs thus are race free.

Theorem 5 (SafeJVML DeadLockFreedom)
Well-typed programs in SafeJVML are free of deadlocks.
Proof: The typing rules for SafeJVML ensure that the lock levels 
in the program form a partial order and that the locks are acquired 
in the decreasing order of their lock levels. The type checking rule 
for acquiring a new lock checks that the level of the lock being ac-
quired is less than \( l_{min} \), which is the topmost value in the \( L_{min} \) se-
quence; the type checking rules guarantee that \( l_{max} \) is the minimum 
level among the levels of all the locks already held in the static lock
set \( \mathcal{L}_S \). The typing rules along with \textit{GoodConfiguration} judgment, 
which ensures the consistency between dynamic and static lock sets
held by the thread, prove that the level of the lock acquired is less 
than the levels of the locks in dynamic lock set \( \mathcal{L}_S \). Thus, well-typed 
programs in SafeJVML are free of deadlocks.

4. Related Work
This section presents work on related type systems. Our type sys-
tem for checking JVML instructions is based on a formalization of 
the JVML type system developed in [20]. Their work covers a large 
subset of JVML but does not handle multithreaded programs.
[4] and [29] provide detailed semantics for JVML but also do not 
handle multithreaded programs. Other formalizations of the JVML 
type system have focused on subroutines [24, 30, 11] and object 
initialization[19]. The type systems in [26, 23] statically verify 
that every method releases all the locks it acquires and no other 
locks. Currently, while this property holds for all well-typed Java 
programs, it does not hold for all well-typed JVML programs that 
pass the bytecode verification. JVMs use runtime checking to en-
sure this property. The type system in [26] is designed for JVML 
programs that are compiled from Java source programs, whereas 
the type system in [23] is more general and supports JVML pro-
grams produced from other sources well. We used ideas from [26]
to track aliases in our system.

None of the previously proposed type systems for JVML handle 
data races, deadlocks, or encapsulation. The main contribution 
of our paper is that, to the best of our knowledge, this is the first type 
system for JVML that statically prevents data races, deadlocks, and 
encapsulation violations.

Our type system extension to JVML is based on a corresponding 
type system extension to Java that we previously developed called 
SafeJava [5, 6, 7, 9]. The SafeJava type system for preventing data 
races is most closely related to [2, 16, 22]. The SafeJava type sys-
tem for enforcing object encapsulation uses a variant of ownership 
types [1, 12, 13, 25]. A detailed comparison of the SafeJava type 
system with related type systems and other approaches for prevent-
ing synchronization errors and encapsulation errors can be found in [5] 
and [6, 7, 9].

References
annotations for program understanding. In Object-Oriented 
Programming, Systems, Languages, and Applications (OOPSLA), 
November 2002.
dialect of Java without data races. In Object-Oriented Programming, 
Systems, Languages, and Applications (OOPSLA), October 2000.
independence, confinement, and access control. In Principles of 
Programming Languages (POPL), January 2002.
[5] Chandrasekhar Boyapati, SafeJava: A unified type system for safe 
programming. Ph.D. thesis, Massachusetts Institute of Technology,
types for safe programming: Preventing data races and deadlocks. In 
Object-Oriented Programming, Systems, Languages, and 
Applications (OOPSLA), November 2002.
Ownership types for object encapsulation. In Principles of 
Programming Languages (POPL), January 2003.
Appendix

A. Rules for Type Checking

This section presents the SafeJVML type system. The SafeJVML grammar is shown in Figure 7. We first define a number of predicates used in the type system. These are based on similar predicates from [17, 16, 5]. We refer the reader to those papers for their precise formulation.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>WFClass(f)</td>
<td>There are no cycles in the class hierarchy</td>
</tr>
<tr>
<td>ClassOnce(f)</td>
<td>No class is declared twice in f</td>
</tr>
<tr>
<td>FieldsOnce(f)</td>
<td>No class contains two fields, declared or inherited, with same name</td>
</tr>
<tr>
<td>MethodsOncePerClass(f)</td>
<td>No class contains two methods with same name</td>
</tr>
<tr>
<td>OverridesOK(f)</td>
<td>Overriding methods have the same return type and parameter types as the methods being overridden. The requirement locks clauses of an overriding method must be superseded by those of the overridden methods</td>
</tr>
<tr>
<td>LockLevelsOK(f)</td>
<td>There are no cycles in the lock levels</td>
</tr>
</tbody>
</table>

We define the typing environment as follows. The typing environment contains the declared types of variables, the declared owner parameters, the declared constraints among owners, and the declared locks clause.

\[ E := \emptyset | E, \text{owner } f | E, \text{constr } E | E, \text{locksclause} \]

We define a minimum lock level as follows:

\[ l_{\text{min}} := \infty | cn.l | \text{LUB}(cn_1.l_1 \ldots cn_k.l_k) \]

By definition, LUB(cn_1.l_1 \ldots cn_k.l_k) &gt; cn_i.l_i \forall i &lt; k. LUB(\ldots) is not computed—it is an expressions used as such for type checking. The lock level \( l_{\text{lock}} \) denotes that the thread currently holds no locks.

We define the type system using the following judgments. We present the typing rules for these judgments after that, except that the typing rules for instructions in methods are given in Figure 11.

<table>
<thead>
<tr>
<th>Judgment</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \vdash P )</td>
<td>Program P is well defined</td>
</tr>
<tr>
<td>( P \vdash \text{defn } f )</td>
<td>defn f is a well-formed class</td>
</tr>
<tr>
<td>( P, E \vdash \text{owner } o )</td>
<td>o is an owner</td>
</tr>
<tr>
<td>( P, E \vdash \text{constr } )</td>
<td>constraint constr is satisfied</td>
</tr>
<tr>
<td>( P, E \vdash t )</td>
<td>t is a well-formed type</td>
</tr>
<tr>
<td>( P, E \vdash t_1 &lt; t_2 )</td>
<td>t_1 is a subtype of t_2</td>
</tr>
<tr>
<td>( P, E \vdash wf E )</td>
<td>typing environment E is well-formed</td>
</tr>
<tr>
<td>( P, E \vdash \text{field } c \in E )</td>
<td>class c declares/inherits field</td>
</tr>
<tr>
<td>( P, E \vdash \text{meth } c \in E )</td>
<td>class c declares/inherits meth</td>
</tr>
<tr>
<td>( P, E \vdash \text{cn.l } c_{n1}.l_1 &lt; c_{n2}.l_2 )</td>
<td>cn_1.l_1 is less than cn_2.l_2 in the partial order formed by lock levels</td>
</tr>
<tr>
<td>( P, E \vdash \text{cn.l } l &lt; l_{\text{min}} )</td>
<td>cn_1.l_1 is less than l_{\text{min}} in the partial order formed by lock levels</td>
</tr>
<tr>
<td>( P, E \vdash \text{Lock}(l) = r )</td>
<td>r is the lock that protects an object of type l</td>
</tr>
<tr>
<td>( P, E, F, S, L, l_{\text{min}} \vdash M )</td>
<td>Given program P, environment E, types of local variables F, types of stack slots S, static lock set LS, and sequence of minimum lock levels L_{\text{min}}, executing the instructions in M does not cause a type error</td>
</tr>
</tbody>
</table>
The above rules define when a JVM program is well-typed. We also define that a heap is well-typed if every record in the heap is well-typed and the runtime state is consistent with the static type information. The function type(h, v) used below is defined in Figure 9.

\[
\text{type}(h, v) = \begin{cases} 
\text{true} & \text{if } h[v] = (f_d, v_i, s_j) \text{ and } f_d \in \text{defn}(h[v_i], v_i) \text{ and } s_j \in \text{defn}(v_i) \\
\text{false} & \text{otherwise}
\end{cases}
\]