The Security of Static Typing with Dynamic Linking

Drew Dean[*]

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Abstract

Dynamic linking is a requirement for portable executable content. Executable content cannot know, ahead of time, where it is going to be executed, nor know the proper operating system interface. This imposes a requirement for dynamic linking. At the same time, we would like languages supporting executable content to be statically typable, for increased efficiency and security. Static typing and dynamic linking interact in a security-relevant way. This interaction is the subject of this paper. One solution is modeled in PVS, and formally proven to be safe.

1 Introduction

When the World Wide Web was composed of static HTML documents, with GIF and JPEG graphics, there was fairly little concern for the security of the browsers. The main objective was to avoid buffer overflow problems which could lead to the execution of arbitrary machine code. When the Web left the research domain and entered the mass market, security became a problem: users wanted electronic commerce. The SSL and S-HTTP protocols were designed to provide cryptographically strong identification of Web servers, and privacy protection for information such as credit card numbers. While an early implementation of SSL had a problem seeding its random number generator [9], and cryptographic protocols are always tricky to design, the situation appeared to be well in hand. Then Java[1][10] arrived. Java has become tremendously popular in 1995-96, primarily due to its support of embedding executable content in World Wide Web pages. Of course, executable content dramatically changes the security of the Web. Java was promoted as addressing the security issue; however, several problems have been found [3].

Java offers a new challenge to computer security: its protection mechanisms are all language-based. Of course, this is really an old idea, but one that has not seen much use since the 1970s. Java is meant to be a "safe" language, where the typing rules of the language provide sufficient protection to serve as the foundation of a secure system. The most important safety property is *type-safety*, by which we mean that a program will never "go wrong" in certain ways: every variable's value will be consistent with the variable's declaration, function calls (i.e., method invocation in the case of Java) will all have the right number and type of arguments, and data-abstraction mechanisms work as documented. All security in Java depends upon these properties being enforced. While the work described here has been inspired by Java, and uses Java concepts and terminology, other systems that base their protection on language mechanisms face similar issues.

One critical issue is the design of dynamic linking[3]. Since Java is a (mostly) statically typed language [10], there exists the potential for a Java applet to run in a different environment than the one in which it was verified, thus leading to a security problem. It was shown that the ability to break Java's type system leads to an attacker being able to run arbitrary machine code, at which point Java can make no security claims[4]. While type theory is a well developed field, there has been relatively little work on the semantics of linking, and less work where linking is a security-critical operation.

This paper addresses the design of a type-safe dynamic linking system. While safe dynamic linking is not sufficient for building a secure system, it is necessary that linking does not break any language properties. The rest of the paper is structured as follows. Section 2 discusses related work, Section 3 gives an informal statement of the problem, Section 4 informally discusses the problem, its ramifications, and solution, Section 5 discusses the formal treatment of the problem in PVS [16], Section 6 briefly discusses implementation and assurance issues, and Section 7 concludes. The PVS specification is provided in an appendix.

2 Related Work

There has been very little recent work in linking. The traditional view is that linkage editing (informally, *linking*, performed by a *linker*) is a static process that replaces symbolic references in object modules with actual machine addresses. The linker takes takes object modules (e.g., Unix[2] `.o` files) produced by a compiler or assembler, along with necessary runtime libraries (e.g., Unix `.a` files) as input, and produces an executable program by laying out the separate pieces in memory, and replacing symbolic references with the machine addresses. Static linking copies code (e.g., the

[*]This work was partially supported by DARPA through Rome Laboratory contract F30602-96-C-0204. Author's present address: Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544, ddean@cs.princeton.edu

[1]Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

[2]Unix is a registered trademark of X/Open, Inc.

standard C library's `printf()` function) and data from the runtime libraries into the executable output. The alternative strategy is dynamic linking.

Although *dynamic* linking is an old idea (appearing in Multics [15], among other systems), it did not become popular in the Unix and PC worlds until the late 1980s to early 1990s, with the advent of systems such as SunOS 4.0 [8] and Microsoft Windows. Dynamic linking delays the replacement of symbolic references with machine addresses until the program is loaded into memory from disk. (In practice, most dynamic linking is *lazy*, that is, a symbolic reference is not replaced until it is used the first time.) Dynamic linking saves both disk space and memory, as there needs to be only one copy of each library on disk, and multiple processes can share the code (assuming it is not self-modifying), but not data areas, in memory. Dynamically-linked programs start up a little slower than statically-linked programs, but this is generally not a problem on modern CPUs.

Besides the memory and disk savings, dynamically linked code offers increased flexibility. Bug fixes in library routines require only the installation of the new libraries, and all dynamically linked programs on the system acquire the fix. Routines with the same interfaces, but different behavior, can be substituted for one another, and the behavior of all dynamically linked programs installed on the system changes.[3] This feature is essential for executable content to be portable. A runtime system abstracts the operating system's system call interface into a portable set of libraries. While the libraries' implementation is platform dependent, all the implementations have the same interface, so the (e.g.) Java applet does not need to know what kind of computer it is running on.

Unix, Macintosh, and PC operating systems, along with C, COBOL, FORTRAN, and Pascal, have treated linking as the process of replacing symbolic references with machine addresses. Since C compilers compile a single file at a time, they cannot detect the same variable being declared differently in different source files. Declaring a variable to be an integer in one file and a pointer in another leads to a unsafe program: trying to interpret an integer as a pointer usually leads to a core dump. Since protection in Java depends on preventing users from forging object references, such a type mismatch would completely undermine the system.

Well-designed languages have module systems that provide support for separate compilation without these problems [21, 13]. C++ introduced *name mangling* as a way to encode type information into linking, to prevent inter-module type errors while still using standard linkers [20]. [4]

Drossopoulou and Eisenbach's recent work[5] considers the type safety of a subset of Java. While it accounts for forward references, it assumes that it is looking at an entire program in a closed world. It does not model the interleaving of type checking, linking, and program execution.

Cardelli's recent work [2] addresses type-safety issues with separate compilation and linking. He introduces a simple language, the simply-typed λ-calculus, with a primitive module system that supports separate compilation. He then informally, but rigorously, proves that his linking algorithm terminates, and if the algorithm is successful, that the resulting program will not have a type error. (Here a type error means calling a function with the wrong number or type(s) of arguments, or using a number as a function.) However, it assumes that all types are known at link time, and does not address (mutually) recursive modules.

Janson's work [11] removes dynamic linking from the Multics kernel. Janson argues that the Multics dynamic linker is *not* security-relevant, so it should not be part of the security kernel. His redesign of dynamic linking moves it into each process, where it happens in user mode rather than kernel mode. (The SunOS 4 dynamic linker design [8] is very similar.) However, dynamic linking in Java *is* security-relevant, unlike Multics, where hardware-based rings were used for protection.

The situation in Java is different from either of the above situations. Java does not have type information available at link time; type checking (that is, byte code verification) is interleaved with dynamic linking. Since the safety of the system relies on type-safety, which in turn relies on proper dynamic linking, the linker is critical to security, unlike the Multics case. This paper considers the security-critical interaction of linking and type checking.

3 Informal Problem Statement

The Java runtime system may interleave type checking, linking, and program execution [10]. The implementation from JavaSoft (and used by Netscape in their Web browser) takes advantage of this freedom. Since most implementations of Java are statically typed, we need to be sure that a linking action cannot invalidate the results of previously performed type checking. If linking could invalidate type checking, then a Java system would be vulnerable to a time-of-check-to-time-of-use (TOCTTOU) attack.

The potential vulnerability is as follows: an applet is downloaded and verified. Part of the verification procedure involves type checking. An applet is (in general) composed of multiple classes, which can reference each other and runtime library components in arbitrary ways (e.g., mutually recursively). The type correctness of the applet depends on the types of these external references to other classes. These classes, if not already present, are loaded during type checking. However, an applet can ask for any arbitrary class to be loaded via a `Class.forName()` call. If a class could load a new class to be used in place of the one it was type checked against, the system would not be type safe. (The actual rules for exactly when Java classes are loaded are very complicated; to make the proofs tractable, we use the simplified system described above.)

The exact correspondence between classes and types is subtle. We use Fisher and Mitchell's model[7], where classes are in 1–1 correspondence with *implementation types*, and implementation types are subtypes of *interface types*, which define the externally visible structure of the class. (Interface types roughly correspond to Java `interface`s.) We say that A is a subtype of B, written $A \leq B$, if an expression of type A can be used in any context where an expression of type B is required. Two implementation types are the same iff they have the same name. (In Java, two classes are the same iff they have the same name and the same classloader[10].) Two interface types are the same if they are structurally equivalent. Interface types fit nicely in the *objects as records* model[1], so we can define structurally equivalent as having the same fields, where corresponding fields have the same type. For an implementation type $Impl$, we write $Impl_{Inter}$ for the corresponding interface type. The interested reader is referred to Fisher's thesis[6] for more details.

We need to define some standard terms from type-theory before we proceed. Let Γ be a *type context* of the form $\Gamma = \{x_1 :$

[3]Hostname lookup in SunOS 4.x is a prime example: the default standard C library provided by Sun uses Sun's NIS to look up hostnames. A system administrator can rebuild the library to use the Internet Domain Name System.

[4]C++ compilers replace function names with symbols that encode the argument and return types of the function. There is no standard algorithm for doing this, which interferes with the interoperability of various C++ compilers on the same machine. This hack was introduced because standard Unix linkers had no way to associate type information with symbols.

$\sigma_1, \ldots, x_k : \sigma_k\}$, where each x_i is a distinct identifier (in this case, they represent classes), and each σ is an implementation type. The notation $x : \sigma$ assigns x the type σ. $\Gamma(x) = \sigma$ iff $x : \sigma \in \Gamma$. Define $x_i \sqsubseteq x_j$ iff $x_{i_{Inter}} = x_{j_{Inter}}$.[5] Define $\Gamma \preceq \Gamma'$ when $\forall x \in \Gamma : \Gamma(x) \sqsubseteq \Gamma'(x)$; we call Γ' a *consistent extension* of Γ.

Let M range over Java classes, which are the objects of type checking. We write $\Gamma \vdash M : \tau$ to mean that M has type τ in context Γ; this is called a *typing judgment.* We assume the following proposition holds:

Proposition 1 *If* $\Gamma \vdash M : \tau$ *and* $\Gamma \preceq \Gamma'$, *then* $\Gamma' \vdash M : \tau$.

The justification for this proposition can be found in [14]; it is a combination of Mitchell's *(add hyp)* axiom and his Lemmas 2.2.1 and 2.2.2. The intuitive reading of this proposition is that we can consistently extend the environment without changing typing judgments in a type system that satisfies the proposition. A rigorous proof of this would require a formalization of the Java type system (see [5] for work in this direction), and is beyond the scope of this paper.

The above definitions are all well and good, but how do they relate to security? Consider a user preparing to run a Java applet embedded in a Web page. Their system provides runtime libraries for the applet, which are under the user's control. The applet's code is completely under its author's control, and was compiled and (hopefully!) tested on his system, against his copy of the runtime libraries. The user's Java runtime implementation may supply additional classes that the author doesn't have. The author would like to know that these will not affect the execution of the applet. The user wants to know that once the applet has been verified (i.e., type checked), that the applet cannot do anything (by adding to or changing its type context) that the verifier would have rejected. Thus, we have a mutual suspicion problem. Under the restrictions given above, the programmer and end-user can safely cooperate.

Restriction 1 (Linking) *A program can only change its type context,* Γ*, to a new type context,* Γ'*, in a way such that* $\Gamma \preceq \Gamma'$.

In summary, by limiting type context modifications to consistent extensions, we can safely perform dynamic linking in the presence of static type checking. The rest of the paper considers the formalization and proof of this statement, along with the consequences of ignoring this limitation.

4 Informal Discussion

The linking restriction given above is a necessary condition so that linking operations do not break the type safety of a language. The designers of Java provided a very flexible dynamic linking facility when they designed the `ClassLoader` mechanism. The basic system only knows how to load and link code from the local file system, and it exports an interface, in the class `ClassLoader`, that allows a Java program to ask the runtime system to turn an array of bytes into a class. The runtime system does not know where the bytes came from; it merely attempts to verify that they represent valid Java byte code. (The byte code is the instruction set of an abstract machine, and is the standard way of transmitting Java classes across the network.) Each class is tagged with the `ClassLoader` that loaded it. Whenever a class needs to resolve a symbolic reference, it asks its own `ClassLoader` to map the name it gives to a class object. Our model always passes the `ClassLoader` as an explicit argument; we prove safety for all `ClassLoader`s.

[5]The reader familiar with object-oriented type theory might expect the definition of $\sqsubseteq$ to be $x_{i_{Inter}} \leq x_{j_{Inter}}$. However, since Java objects are really object references, and the Java class hierarchy is acyclic (i.e., $\leq$ is a partial order, not just a pre-order) there is no statically sound subtype relation other than equality.

The original Java Development Kit (JDK) implementation (JDK 1.0.2) did not place any restrictions on the behavior of `ClassLoader`s. This led to the complete breakage of type safety, where integers could be used as object references, and vice versa [3]. The type safety failure led to an untrusted applet being able to run arbitrary machine code, thus completely compromising the security of Java applets [4]. After discussion with Sun, language was added to the definition of Java [10] restricting `ClassLoader`s to safe behavior. Code to implement this restriction (essentially the same as the linking restriction) has not yet shipped, but is expected shortly in JDK 1.1.

The absence of the linking restriction directly led to two problems in the JDK 1.0.2 implementation:

1. A rogue `ClassLoader` can break the semantics of Java by supplying inconsistent mappings from names to classes. In earlier JDK releases, and Netscape Navigator 2.0x, this led to complete compromise of the system.
2. Another bug was found in JDK 1.0.2's handling of array classes. (In Java, all arrays are objects, and suitable class definitions are automatically generated.) It was possible to trick the system into loading a user-defined array class while the program was running, aliasing a memory location as both an object reference and an integer. The static type checking was performed against the real array class, and then the program loaded the bogus array class by its request, which was an not a consistent extension of the type context. This bug was in the `AppletClassLoader` supplied by Sun, and exploitable by web applets. This also led to running arbitrary machine code, completely compromising the security of the system.

The PVS specification presented below specifies a simple implementation of dynamic linking. It restricts linking to consistent extensions of the current type context. It shows that all relevant operations invariantly preserve consistency of the type context. It proves that the initial context (here, a cut down version of the Java runtime library) is consistent. The combination of these properties is an inductive proof of the safety of the system.

5 Formal Treatment in PVS

PVS[16][6] is the PROTOTYPE VERIFICATION SYSTEM, the current SRI research project in formal methods and theorem proving. PVS has been used to verify many different projects, including a microprocessor[19], floating point division[18], fault-tolerant algorithms[12], and multimedia frameworks[17], by users at SRI and other sites. PVS combines a specification language with a variety of theorem proving tools.

Proposition 1 states that security is preserved if a program is linked and run in a consistent extension of the type context it was compiled in. Any actual implementation of dynamic linking will be quite complex, and it is not obvious that a particular implementation satisfies Proposition 1. This paper builds a model of dynamic linking that is quite similar to the Java implementation, and proves that this model ensures type-safety. By writing a concrete specification in PVS, and proving the desired properties, we get a

[6]For more information about PVS, see `http://www.csl.sri.com/pvs.html`

specification that looks very much like a functional program, along with a correctness proof. While some specification writers would prefer a more abstract specification (with key properties defined as axioms, and many functions unspecified), we chose to give a very concrete specification, to make it easier to relate to an actual implementation. PVS's proof facilities are strong enough to make this specification verifiable without undue difficulty.

5.1 The PVS Model

It should be noted that the model is fairly closely related to how Sun's Java implementation performs dynamic linking, but it is *not* a model of Java. Certain simplifications were made to Java, and the model fixed design problems observed in the JDK 1.0.2 implementation. Sun has been working on their system as well, and coincidentally certain features are similar, but these are independent designs, and one should be careful not to confuse the results of this paper with any products. This model merely shows that dynamic linking can peacefully co-exist with static typing.

5.1.1 PVS Types

The core structure in the model is the `ClassTable`, which contains two mappings: the first, an *environment* mapping `(Name, ClassLoader)` pairs to `ClassID`s, and the second, a *store* mapping `ClassID`s to `Class` objects. The terms "environment" and "store" are meant to reflect similar structures in programming language semantics. The environment associates names with locations (on a physical machine, memory addresses), and the store simulates RAM. The indirection between `(Name,ClassLoader)` pairs and `Class`es exists so that linking does not have to change the environment; it only changes the store. This allows us to show that the environment does not change over time, even if the actual objects that the names are bound to do. Note that we keep a mapping from a `(Name,ClassLoader)` pair to a list of `ClassID`s; the correctness proof is that there is at most one `ClassID` associated with each name, i.e., that this mapping is a partial function. We keep a list of `ClassID`s instead of a set, so we can tell what order things happened in if anything should ever break. We define a state as safe iff each `(Name, ClassLoader)` pair maps to at most one `ClassID`.[7]

We declare `ClassLoader` to be an uninterpreted type with at least one element. The natural model of the Java `ClassLoader` would be a mutually recursive datatype with `Class`, but PVS does not handle the mutual recursion found in the Java implementation conveniently. Since our model only uses the `ClassLoader` as part of the key in the `ClassTable`, it suffices for `ClassLoader` to be uninterpreted.

The `Class` datatype represents classes in our model. A class has either been resolved (i.e., linked), or unresolved, in which case the class has no pointers to other classes, but only unresolved symbols. One might be tempted to use only the resolved constructor, but PVS requires that each datatype have a non-recursive constructor.

The `ClassID` is imported from the `identifiers` theory. These are merely unique identifiers; currently they are implemented in the obvious fashion using integers. It is better to define a theory for identifiers, so that other representations can be used later, without changing the proofs. The `ClassIDMap` plays the role of a store in semantics, giving a mapping between `ClassID`s and `Class`es. `ClassDB` is a pair consisting of the next unused identifier, and a `ClassIDMap`.

We represent objects by the type `Object`, which merely records which class this object is an instance of. While this representation is fairly abstract, it suffices for our proofs.

[7]The model is defined in a way such that the set of `(Name, ClassLoader)` to `ClassID` mappings is monotonically increasing. This property makes the safety definition sufficient. However, a formal proof that the mapping is time-invariant would be nice. This is future work.

5.1.2 PVS Implementation

The structure of our model roughly follows Sun's Java Virtual Machine implementation. The major exception is that PVS does not have global variables or mutation, so we explicitly pass the state of the system to each function. We have also rearranged some data structures for ease in modeling.

Primitive Operations

The `FindClassIDs` function takes a `ClassTable`, the name of a class, and the requested `ClassLoader`, and returns a list of `ClassID`s. `FindClass` applies the current store, mapping `ClassID`s to `Class`es, to the result of `FindClassIDs`.

The `InsertClass` function takes a `ClassTable`, the name and ClassLoader of a new class, and the new class, and inserts it into the `ClassTable`. It returns the new `ClassTable`. Note that the insertion generates a new `ClassTable` — it does not destroy the old one. This is a low-level utility function that does not enforce any invariants; those are supplied at a higher level.

The `ReplaceClass` function takes a `ClassTable`, the old and new classes, and the appropriate `ClassLoader`, and updates the store iff the appropriate class is found. It then returns the new `ClassTable`. If no appropriate class is found, it returns the unchanged `ClassTable`.

Class Loading

The `define` function is modeled after the Java `defineClass()` function. It takes a `ClassTable`, the name of the new class, the unresolved references of the new class, and a `ClassLoader`. It returns a pair: the new class and the updated class table. No invariants are checked at this level. This corresponds to the Java design, where `defineClass()` is a protected method in `ClassLoader`, and is only called after the appropriate safety checks have been made.

The `loadClass` function plays a role similar to `loadClass()` in a properly operating Java `ClassLoader`. In the Java system, `loadClass()` is the method the runtime system uses to request that a `ClassLoader` provide a mapping from a name to a `Class` object. Our model checks whether the class is provided by the "runtime system," by checking the result of `findSysClass`. We then check whether this `ClassLoader` has defined the class, and return it if so. Otherwise, we define a new class. Since this class could come from anywhere, and contain anything (we assume only valid classes), we tell PVS that some external references exist in the `Input: (cons?[string])` construction, without specifying any particular external references.

The `linkClass` function, although it plays a supporting role, is defined here because PVS does not allow forward references. The `linkClass` function takes a `ClassTable`, the class to be linked, and the class's `ClassLoader`, and returns the linked class, and the updated `ClassTable`. The linking algorithm is very simple: while there is an unresolved reference, find the class it refers to, (loading it if necessary, which could create a new

ClassTable), and resolve the reference. The linkClass function only returns "resolved" classes; these may be partially resolved in the recursive calls to linkClass during the linking process.

The resolve function is modeled after the Java resolveClass() method. It takes a ClassTable, class, and class loader, links the class with respect to the given ClassLoader, and updates the ClassTable. It returns the new ClassTable.

Classes Classes have several operations: the ability to create a new instance of the class, ask the name of a class, get a class's ClassLoader, and to load a new class. Loading a new class is the only non-trivial operation; it simply invokes loadClass.

The Java runtime system provides several classes that are "special" in some sense: java.lang.Object is the root of the class hierarchy, java.lang.Class is the class of Class objects, and java.lang.ClassLoader defines the dynamic linking primitives. These classes play important roles in the system; we model this behavior by assuming they are pre-loaded at startup.

5.2 The Proofs

This paper offers two contributions: While Proposition 1 is a simple statement, it is a necessary restriction whose importance has been overlooked, especially in the initial design and implementation of Java. The concept, though, is generic: any language whose type system satisfies Proposition 1 (and most do) can use the results of this paper. Given an operational semantics for the language under inspection, a completely formal safety proof can be constructed. Drossopoulou and Eisenbach's work[5] is a good beginning, but was not available when this work began. The second contribution is a proof that the requirements of Proposition 1 are satisfied by our model. Here the proofs are discussed at a high level; PVS takes care of the details.

There are three lemmas, two *putative theorems*, labeled as conjectures, and five theorems which establish the result. The putative theorems are checks that the specification conveys the intent of the author. Formal proof of these theorems increases our confidence in the correctness of the specification. The five theorems show that the system starts operation in a safe state, and each operation takes the system from a safe state to a safe state. Since the theorems are universally quantified over class names, classloaders, classes, and class tables, any interleaving of the functions (assuming each function is an atomic unit) is safe. All of the theorems have been formally proven in PVS; here we only present brief outlines of the proofs. The details are all routine, and taken care of by PVS.

5.2.1 Lemmas

MapPreservesLength Map is a function that takes a function and a list, and returns the list that results from applying the function to each element of the list.[8] MapPreservesLength simply asserts that the length of the resulting list equals the length of the argument list. The proof is by induction on the length of the list and the definition of map.

[8]Map is a standard function in most functional programming languages. While the standard PVS definition is slightly complicated, it is equivalent to:
```
map(l:  list[T], f:  function[T -> S]) : RECURSIVE
list[S] = IF null?[l] THEN null ELSE cons(f(car(l)),
map(cdr(l), f)) ENDIF
```

proj1_FindClassIDs This lemma asserts the independence of the environment, mapping (Name, ClassLoader) pairs to ClassID lists, and the store, mapping ClassIDs to Classes. The lemma states that for all ClassTables, looking up a name in the environment gives the same result no matter what store is supplied. The proof is by induction on the size of the environment. It's clearly true for the empty environment, and the store is not referenced during the examination of each binding.

safe_proj This technical lemma is needed in the proof of resolve_inv. It states that a safe ClassTable is still safe when its store is replaced by an arbitrary store. Since safety is a function of the environment, not the store, this is intuitively obvious. The proof uses the MapPreservesLength lemma.

5.2.2 Conjectures

Add This putative theorem was the first one proven, to check our understanding of the specification. It states that looking up a class, after inserting it, returns at least one class. PVS automatically proves this theorem.

Resolve This putative theorem states that linking terminates by producing a class with no unresolved references. (We do not model the failure to find an unresolved reference.) The proof is by induction on the number of unresolved references. Clearly it holds for a completely resolved class, and each recursive call to linkClass resolves one class reference.

5.2.3 Theorems

forName_inv This is the first case of the invariant. It states that the forName function preserves safety. The proof follows from the lemmas MapPreservesLength and proj1_FindClassIDs.

Initial_Safe This theorem states that the system initially starts out in a safe state. With the aid of the string_lemmas theory, written by Sam Owre, PVS proves this theorem automatically. Since the initial state has finite size, the safety property is very simple to check.

loadClass_inv This is the next case to consider in proving the invariant. It states that the loadClass function is safe, in the sense that it will never bind a (Name, ClassLoader) pair to a Class if such a binding already exists. The proof is very similar to forName_inv.

linkClass_inv This case of the invariant states that linkClass preserves safety. The intuitive idea is that linkClass only modifies the store, not the environment. The proof is fairly complicated, using loadClass_inv as a lemma, proceeds by induction on the number of unresolved references in the class.

resolve_inv This is the last case of the invariant. It states that the resolve operation is safe. This is intuitively obvious, since resolve is the composition of linkClass and ReplaceClass, neither of which modifies the environment. The proof uses linkClass_inv as a lemma, and then does a case split on the result of FindClassIDs. If FindClassIDs returns a list, the safe_proj lemma leads to the desired result. If FindClassIDs returns null, the result is immediate.

6 Implementation and Assurance

This paper has discussed a *model* of dynamic linking, and proven a safety property under one assumption. While this is a nice result, systems in the real world get implemented by humans. A couple of simplifications were made with respect to Java:

1. Class names were assumed to be in canonical form; Java requires mapping "." to "/" at some point. Since this is not a 1–1 correspondence, it needs to be handled consistently.

2. The fact that array classes (classes with names beginning with a [) have a special form has not been modeled.

3. The failure to locate a class is not modeled. We assume that such a failure will halt program execution, via an unspecified mechanism.

The basic conclusion for implementors is that each class definition must be loaded *exactly* once for each classloader. The simplest way to do this is for the runtime system to track which classes have been loaded by which classloaders and only ask a classloader to provide the definition of a class once. We assume that a classloader will either provide a class or fail consistently.

The assurance level of the final system will depend on many factors. We note that our mechanism is conceptually simple, and can be specified in three pages. Our proofs were performed with lists, because they are simple to do inductive proofs on. A real implementation would probably use a more efficient data structure. However, it should be simple to show that other data structures, e.g., a hash table, satisfy the required properties. The specification contains no axioms, and is essentially a functional program, in the sense that it shows exactly what is to be computed, and so could serve as a prototype implementation. Clearly, though, dynamic linking is part of the trusted computing base for Java and similar systems, and a given system will have an assurance level no higher than the assurance of its dynamic linking.

7 Conclusion

This paper presents one of many models for dynamic linking. A formal proof is presented to show that dynamic linking need not interfere with static type checking. While the system presented is not Java, it is closely related, and can serve as a proof-of-concept for Java implementors. Studying the JDK implementation for the purpose of modeling it for this work led to the discovery of a type-system failure in JDK 1.0.2 and Netscape Navigator 2.02. The proofs presented here were not unduly hard to generate, and greatly improve confidence in the safety of dynamic linking.

8 Acknowledgments

The work reported on in this paper was done while the author was visiting the Computer Science Laboratory at SRI International. The visit was arranged by Peter Neumann and John Rushby. Technical assistance, without which this work would not have been possible, was provided by David Stringer-Calvert, Natarajan Shankar, and Sam Owre. The content and presentation of this work were greatly enhanced by comments from Andrew Appel (Princeton University), Ed Felten (Princeton University), Peter Neumann, John Rushby, and Natarajan Shankar. I would also like thank all the other members of the laboratory who made my stay a very pleasant and productive experience.

References

[1] CARDELLI, L. A semantics of multiple inheritance. *Information and Computation 76* (1988), 138–164.

[2] CARDELLI, L. Program fragments, linking, and modularization. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages* (Jan. 1997). To appear.

[3] DEAN, D., FELTEN, E. W., AND WALLACH, D. S. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (May 1996), pp. 190–200.

[4] DEAN, D., FELTEN, E. W., AND WALLACH, D. S. Java security: From HotJava to Netscape and beyond. In *Computers Under Attack*, P. Denning, Ed., 2nd ed. ACM Press, 1997. To appear.

[5] DROSSOPOULOU, S., AND EISENBACH, S. Is the Java type system sound? In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages* (Paris, Jan. 1997). To appear.

[6] FISHER, K. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996.

[7] FISHER, K., AND MITCHELL, J. C. On the relationship between classes, objects, and data abstraction. In *Proceedings of the 17th International Summer School on Mathematics of Program Construction* (Marktoberdorf, Germany, 1996), LNCS, Springer-Verlag. To appear.

[8] GINGELL, R. A., LEE, M., DANG, X. T., AND WEEKS, M. S. Shared libraries in SunOS. In *USENIX Conference Proceedings* (Phoenix, AZ, 1987), pp. 131–145.

[9] GOLDBERG, I., AND WAGNER, D. Randomness and the netscape browser. *Dr. Dobb's Journal* (Jan. 1996).

[10] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. Addison-Wesley, 1996.

[11] JANSON, P. A. Removing the dynamic linker from the security kernel of a computing utility. Master's thesis, Massachusetts Institute of Technology, June 1974. Project MAC TR-132.

[12] LINCOLN, P., AND RUSHBY, J. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In *Computer-Aided Verification, CAV '93* (Elounda, Greece, June/July 1993), C. Courcoubetis, Ed., vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 292–304.

[13] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[14] MITCHELL, J. C. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B: Formal Models and Semantics. Elsevier Science Publishers B.V., 1990, ch. 8.

[15] ORGANICK, E. *The Multics System: An Examination of its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[16] OWRE, S., SHANKAR, N., AND RUSHBY, J. M. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Three volumes: Language, System, and Prover Reference Manuals; A new edition for PVS Version 2 is expected in late 1996.

[17] RAJAN, S., RANGAN, P. V., AND VIN, H. M. A formal basis for structured multimedia collaborations. In *Proceedings of the 2nd IEEE International Conference on Multimedia Computing and Systems* (Washington, DC, May 1995), IEEE Computer Society, pp. 194–201.

[18] RUESS, H., SHANKAR, N., AND SRIVAS, M. K. Modular verification of SRT division. In *Computer-Aided Verification, CAV '96* (New Brunswick, NJ, July/August 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 123–134.

[19] SRIVAS, M. K., AND MILLER, S. P. Formal verification of the AAMP5 microprocessor. In *Applications of Formal Methods*, M. G. Hinchey and J. P. Bowen, Eds., Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1995, ch. 7, pp. 125–180.

[20] STROUSTRUP, B. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[21] WIRTH, N. *Programming in Modula-2*, 2nd ed. Springer-Verlag, 1983.

A The PVS Specification

The PVS specification language builds on a classical typed higher-order logic. The base types consist of booleans, real numbers, rationals, integers, natural numbers, lists, and so forth. The primitive type constructors include those for forming function (e.g., `[nat -> nat]`), record (e.g., `[# a : nat, b : list[nat]#]`) , and tuple types (e.g., `[int, list[nat]]`). PVS terms include constants, variables, abstractions (e.g., `(LAMBDA (i : nat): i * i)`), applications (e.g., `mod(i, 5)`), record constructions (e.g., `(# a := 2, b := cons(1, null) #)`), tuple constructions (e.g., `(-5, cons(1, null))`), function updates (e.g., `f WITH [(2) := 7]`), and record updates (e.g., `r WITH [a := 5, b := cons(3, b(r))]`). Note that the application `a(r)` is used to access the `a` field of record `r`, and the application `PROJ_2(t)` is used to access the second component of a tuple `t`. PVS specifications are packaged as *theories*.

```
Types : THEORY
  BEGIN

  IMPORTING string_lemmas, identifiers

  ClassLoader : TYPE+

  Class : DATATYPE
    BEGIN
    resolved(name : string, references : list[string], loader : ClassLoader, linked : list[Class]) :
      resolved?
    unresolved(name : string, references : list[string], loader : ClassLoader) : unresolved?
    END Class


  ClassID : TYPE = Ident

  ClassList : TYPE = list[Class]

  ClassIDMap : TYPE = FUNCTION[ClassID → Class]

  ClassDB : TYPE = [ClassID, ClassIDMap]

  ClassTable : TYPE = [list[[string, ClassLoader, list[ClassID]]], ClassDB]

  Object : TYPE+ = [# cl : Class#]

  primordialClassLoader : ClassLoader

  mkClass((nm : string), (refs : list[string]), (ldr : ClassLoader)) :
    Class = unresolved(nm, refs, ldr)

  bogusClass : Class = mkClass( ”” , null, primordialClassLoader)

  emptyClassTable : ClassTable = (null, (initialID, λ (id : ClassID) : bogusClass))

  FindClassIDs((ct : ClassTable), (nm : string), (cldr : ClassLoader)) :
    RECURSIVE list[ClassID] = CASES PROJ_1(ct) OF
      null : null,
      cons(hd, tl) :
        LET tab = PROJ_1(ct), db = PROJ_2(ct)
          IN IF
              PROJ_1(hd) = nm∧
                PROJ_2(hd) =
                  cldr
             THEN PROJ_3(hd)
            ELSE
             FindClassIDs((tl, db), nm, cldr)
            ENDIF
      ENDCASES
    MEASURE length(PROJ_1(ct))

  FindClass((ct : ClassTable), (nm : string), (cldr : ClassLoader)) :
    ClassList = map(PROJ_2(PROJ_2(ct)), FindClassIDs(ct, nm, cldr))

  InsertClass((ct : ClassTable), (nm : string), (cldr : ClassLoader), (cl : Class)) : ClassTable =
    LET old = FindClassIDs(ct, nm, cldr),
        newID = GetNextID(PROJ_1(PROJ_2(ct))),
        newMap = PROJ_2(PROJ_2(ct)) WITH [newID := cl]
    IN (cons((nm, cldr, cons(newID, old)), PROJ_1(ct)), (newID, newMap));

  ReplaceClass((ct : ClassTable), (cl, newCl : Class), (cldr : ClassLoader)) : ClassTable =
    LET classDB = PROJ_2(PROJ_2(ct)),
        id = PROJ_1(PROJ_2(ct)),
        tab = PROJ_1(ct),
        clID = FindClassIDs(ct, name(cl), cldr)
    IN CASES clID OF cons(hd, tl) : (tab, (id, classDBWITH [hd := newCl])), null : ctENDCASES

  define((ct : ClassTable), (nm : string), (refs : list[string]), (cldr : ClassLoader)) :
    [Class, ClassTable] = LET cl = mkClass(nm, refs, cldr) IN (cl, InsertClass(ct, nm, cldr, cl))

  findSysClass((ct : ClassTable), (nm : string)) :
    ClassList = FindClass(ct, nm, primordialClassLoader)

  foo : list[string] = cons( "foo" , null)

  Input : (cons?[string])

  loadClass((ct : ClassTable), (nm : string), (cldr : ClassLoader)) : [Class, ClassTable] =
    LET local = findSysClass(ct, nm), loaded = FindClass(ct, nm, cldr)
    IN IF null?(local) THEN IF cons?(loaded) THEN (car(loaded), ct)
         ELSE define(ct, nm, Input, cldr)
         ENDIF
        ELSE (car(local), ct)
        ENDIF;

  linkClass((ct : ClassTable), (cl : Class), (cldr : ClassLoader)) :
    RECURSIVE [Class, ClassTable] = LET getClass = (λ (n : string) : loadClass(ct, n, cldr))
      IN CASES references(cl) OF
        null :
            IF unresolved?(cl)
             THEN (resolved(name(cl), null, loader(cl), null),
                   ct)
            ELSE (cl, ct)
            ENDIF,
        cons(hd, tl) :
            LET (res, newCt) = getClass(hd),
                newCl = CASES cl OF
                             unresolved(name,
                             references,
                             loader) :
                                resolved(name, tl,
                                         loader,
                                         cons(res, null)),
                             resolved(name,
                             references,
                             loader, linked) :
                                resolved(name, tl,
                                         loader,
                                         cons(res, linked))
                             ENDCASES
             IN linkClass(newCt, newCl, cldr)
        ENDCASES
    MEASURE length(references(cl))

  resolve((ct : ClassTable), (cl : Class), (cldr : ClassLoader)) : ClassTable =
    LET (newCl, newCt) = linkClass(ct, cl, cldr) IN ReplaceClass(newCt, cl, newCl, cldr);

  forName((ct : ClassTable), (nm : string), (cldr : ClassLoader)) : [Class, ClassTable] =
    CASES FindClass(ct, nm, cldr) OF cons(hd, tl) : (hd, ct), null : loadClass(ct, nm, cldr) ENDCASES

  newInstance((clss : Class)) : Object = (#cl := clss#)

  getClassLoader((cl : Class)) : ClassLoader = loader(cl)

  getName((cl : Class)) : string = name(cl)

  jlObjectClass : Class =
    mkClass( "java.lang.Object" , null, primordialClassLoader)

  jlClassClass : Class =
    mkClass( "java.lang.Class" ,
            cons( "java.lang.Object" , null), primordialClassLoader)

  jlClassLoaderClass : Class =
    mkClass( "java.lang.ClassLoader" ,
            cons( "java.lang.Object" ,
                  cons( "java.lang.Class" , null)),
            primordialClassLoader)

  sysClassTable : ClassTable =
    InsertClass(InsertClass(InsertClass(emptyClassTable,
                                        "java.lang.Object" ,
                                        primordialClassLoader,
                                        jlObjectClass),
          "java.lang.Class" ,
          primordialClassLoader, jlClassClass),
        "java.lang.ClassLoader" ,
        primordialClassLoader, jlClassLoaderClass)

  ct : VAR ClassTable

  nm : VAR string

  cldr : VAR ClassLoader

  cl : VAR Class

  MapPreservesLength : LEMMA
      (∀ (f : FUNCTION[ClassID → Class]), (l : list[ClassID]) :
        length(map(f, l)) = length(l))

  proj1_FindClassIDs : LEMMA
      (∀ (ct : ClassTable), (nm : string), (cldr : ClassLoader), (classdb : ClassDB) :
        FindClassIDs((PROJ_1(ct), classdb), nm, cldr) = FindClassIDs(ct, nm, cldr))

  Add : CONJECTURE
      (∃ (cll : ClassList) :
        FindClass(InsertClass(ct, nm, cldr, cl), nm, cldr) = cons(cl, cll))

  Resolve : CONJECTURE
      (∀ (cl : Class), (ct : ClassTable), (cldr : ClassLoader) :
        references(PROJ_1(linkClass(ct, cl, cldr))) = null)

  Safe((ct : ClassTable)) : bool =
    (∀ (nm : string), (cldr : ClassLoader) :
        LET cll = length(FindClass(ct, nm, cldr)) IN cll ≤ 1)

  safe_proj : LEMMA
      (∀ ct, (mapping : ClassIDMap) :
        Safe(ct) ⊃ Safe(PROJ_1(ct), (PROJ_1(PROJ_2(ct)), mapping)))

  forName_inv : THEOREM (∀ ct, nm, cldr : Safe(ct) ⊃ Safe(PROJ_2(forName(ct, nm, cldr))))

  Initial_Safe : THEOREM Safe(sysClassTable)

  loadClass_inv : THEOREM
      (∀ ct, nm, cldr : Safe(ct) ⊃ Safe(PROJ_2(loadClass(ct, nm, cldr))))

  linkClass_inv : THEOREM
      (∀ ct, cl, cldr : Safe(ct) ⊃ Safe(PROJ_2(linkClass(ct, cl, cldr))))

  resolve_inv : THEOREM (∀ ct, cl, cldr : Safe(ct) ⊃ Safe(resolve(ct, cl, cldr)))

  END Types
```