Formal Aspects of Mobile Code Security

Richard Drews Dean

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 1999

Abstract

We believe that formal methods of all kinds are critical to mobile code security, as one route to gaining the assurance level necessary for running potentially hostile code on a routine basis. We begin by examining Java, and understanding the weaknesses in its architecture, on both design and implementation levels. Identifying dynamic linking as a key problem, we produce a formal model of linking, and prove desirable properties about our model. This investigation leads to a deep understanding of the underlying problem. Finally, we turn our attention to cryptographic hash functions, and their analysis with binary decision diagrams (BDDs). We show that three commonly used hash functions (MD4, MD5, and SHA-1) do not offer ideal strength against second preimages. The ability of a cryptographic hash function to resist the finding of second preimages is critical for its use in digital signature schemes: a second preimage enables the forgery of digital signatures, which would undermine confidence in digitally signed mobile code. Our results show that modern theorem provers and BDD-based reasoning tools are effective for reasoning about some of the key problems facing mobile code security today.

Acknowledgments

My advisor, Andrew Appel, offered sure guidance through what turned out to be an exceptionally smooth journey through graduate school. I can only offer my deepest gratitude. Ed Felten freely offered his time and advice, even though I was not his student. Several of those late-night hacking sessions will remain forever unforgettable.

To my long suffering office mates, both past and present, I offer thanks for putting up with me for four years. Four years can be either the shortest or longest periods of time; Martin Carlisle, Steven Kleinstein, Paul Martino, Don Mitchell, Anil Ruia, and Michael Schneider all deserve thanks for their support. Jeff Korn and Dan Wang provided valuable friendship and technical insights; I can only hope they will say the same when their turn comes. Paul Burchard helped me enjoy many trips to Small World. Dan Boneh and Pei Cao provided collegial atmosphere when I arrived, and I thank Pei for introducing me to PARC. Last, but not least, I've had an amazing collaboration with Dan Wallach, as well as good sushi. Thanks. The folks known as "CS Staff" deserve a special thanks, for keeping things working, even in the presence of some not entirely reasonable requests.

I've been privileged to meet lots of helpful people outside Princeton. The kind and generous people at the SRI International Computer Science Laboratory have helped make this thesis possible with two summers of support, both financial and technical. I thank Li Gong, John Rushby, Patrick Lincoln, Sam Owre, Natarajan Shankar, Paul Jackson, and David Stringer-Calvert. Peter Neumann deserves special thanks, for also agreeing to serve on my thesis committee.

Others who I owe technical debts to include Martín Abadi (DEC), Kathleen Fisher (AT&T), Paul Karger (IBM), John Mitchell (Stanford), Jon Riecke (Bell Labs), and Andrew Wright (Intertrust). Jim Roskind (Netscape) always took my phone calls with good grace, even when they generated lots of work for him. Kim Bruce, Frank Pfenning, and Dan Wang provided good technical advice for dotting the i's and crossing the t's in Chapter 4. All errors, of course, are the fault of the author.

I thank Alan Hu for long-term friendship, and recent collaboration; the results in Chapter 5 would be absent without his expert advice. I thank Ray Chiang for long-term friendship, even with continents and oceans in the way.

Research in the Secure Internet Programming Laboratory at Princeton is supported by the NSF under award CCR-9457813, by the Alfred P. Sloan Foundation, and by donations from Sun Microsystems, Microsoft, Intel, Merrill Lynch and Trintech. My work at SRI was partially supported by DARPA through Rome Laboratory contract F30602-96-C-0204.

To my parents

Contents

Abstract iii

1 Introduction **1**

2 Java Security[1] **6**

2.1 Introduction 6

2.2 Java Semantics 10

2.3 Java Security Mechanisms 14

2.4 Taxonomy 15

2.4.1 Denial-of-Service Attacks 15

2.4.2 Confidentiality Breaches 16

2.4.3 Implementation Errors 20

2.4.4 Interapplet Security 24

2.4.5 Java Language Implementation Failures 25

2.5 Java Language and Bytecode Weaknesses 39

2.5.1 Language Weaknesses 40

[1]Earlier versions of this chapter appeared in the *1996 IEEE Symposium on Security and Privacy* [DFW96] and *Internet Besieged: Countering the Cyberspace Scofflaw* [DFWB97].

2.5.2 Bytecode Weaknesses 41

2.6 Analysis 44

2.6.1 Policy 45

2.6.2 Enforcement 45

2.6.3 Integrity 46

2.6.4 Accountability 47

2.7 Conclusion 48

3 Safe Dynamic Linking[2] **49**

3.1 Introduction 49

3.2 Related Work 50

3.3 Informal Problem Statement 53

3.4 Informal Discussion 57

3.5 Formal Treatment in PVS 61

3.5.1 The PVS Model 61

3.5.2 The Proofs 67

3.6 Implementation and Assurance 81

3.7 Conclusion 83

4 Linking and Scope **84**

4.1 Related Work 85

4.2 Modeling the Problem in PCF 86

[2]An earlier version of this chapter appeared in the *Fourth ACM Conference on Computer and Communication Security* [Dea97].

4.3 PCF$_\tau$ 87
4.4 PCF$_{\tau,\rho}$ 89
4.4.1 PCF$_{\tau,\rho}$ Definition 89
4.4.2 PCF$_{\tau,\rho}$ Operational Semantics 91
4.4.3 PCF$_{\tau,\rho}$: The Type System 91
4.4.4 PCF$_{\tau,\rho}$ Domains 94
4.4.5 PCF$_{\tau,\rho}$ Environments 94
4.4.6 PCF$_{\tau,\rho}$ Example 101
4.5 Solution for PCF$_{\tau,\rho}$ 102
4.5.1 Formalization 103
4.6 Relating PCF$_{\tau,\rho}$ to Java 111
4.7 Conclusions 115

5 Attacking Cryptographic Hash Functions **116**

5.1 Introduction 116
5.2 Notation 117
5.3 Properties and Attacks 119
5.3.1 Second Preimage Attack 120
5.4 Binary Decision Diagrams[3] 126
5.5 Specifying MD4, MD5 and SHA-1 in Ever 128
5.6 Defenses 129
5.6.1 Strengthening MD Strengthening 130

[3]This presentation closely follows Hu [Hu97].

5.6.2 Combining Function 131

5.6.3 Multiple Hashing 132

5.7 Related Work 133

5.8 Conclusion 134

6 Conclusions 135

A PVS Specification 140

B Formal Proofs 148

Chapter 1

Introduction

The work reported on in this thesis began in 1995. Sun Microsystems had recently released the Java[1] programming language, and marketed it as a portable, safe, and secure way to attach small programs, called *applets*, to Web pages. The World Wide Web was still new and novel at the time; version 1 of Netscape Navigator had been available for about 6 months. Demand for new features was intense, but HTML (like all standards) was evolving slowly. The ability to embed arbitrary programs in Web pages would allow content providers great freedom to innovate. It would also solve network bandwidth and latency problems, by running the program locally on the user's machine. Of course, the idea of attaching programs to Web pages has massive security implications; users should not have to consider the security consequences of browsing a Web page.

To show that Java was a real language, Sun produced a Web browser, HotJava, itself written in Java. The alpha release of HotJava was the first browser to support applets. Netscape licensed Java in summer 1995, and shipped Java in version 2

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

of Navigator in spring 1996. Microsoft licensed Java in the winter of 1995, and shipped it in version 3 of Internet Explorer. Sun did not follow up with a 1.0 release of HotJava until spring 1997. By this time, Netscape and Microsoft had effectively captured the market for Web browsers. Although Web browsers have been the most widespread deployment of Java, one should not consider Java embedded in a browser to be the whole story.

Java became the first type-safe, garbage-collected language to enter the mainstream of the computing industry. Although there have been many safe languages developed in the research community (various Lisp dialects, ML, Mesa, etc.), none of them have made much impact on mainstream computing. This is too bad, because applications developed in safe languages should be more robust due to the lack of memory corruption errors. For security-critical applications, making it impossible to implement a buffer overflow is a fabulous benefit[2].

Using a safe language (that has some support for maintaining abstractions) potentially offers a high-performance, portable way of achieving *memory safety*, *i.e.,* the ability to control what memory locations a program may read and/or write. Although memory safety is not absolutely necessary for security in all settings[3], it is necessary for security in the absence of detailed analysis of the programs running concurrently. If we consider the virtual memory hardware of the CPU to be an interpreter, guaranteeing memory safety with dynamic checks, the first question we ask when wearing a programming language hat is "How can we turn those dynamic checks into static checks?" A statically type-safe language with ab-

[2]See the continuing saga of buffer overflows on the BUGTRAQ mailing list (archives available from `http://www.netspace.org/lsv-archive/bugtraq.html`). Even though it is a 30 year old problem, all too many privileged programs written in C are vulnerable.

[3]Consider two programs, A and B, running concurrently. A owns locations x_1 and x_2; B owns locations y_1 and y_2. Suppose (for known, fixed, A and B), at some point in time, x_2 and y_2 become dead variables. After that, B can overwrite x_2, and A can overwrite y_2. If our definition of security is non-interference, the system is secure.

straction mechanisms is one answer to the memory safety problem; proof-carrying code [NL96] is another.

Once the memory safety problem has been solved, the remaining problem is to provide useful, yet secure, interfaces to the outside world. The class libraries that come with Java present a set of classes to the programmer for interacting with the non-Java world. These classes use Java's abstraction mechanisms to force their callers to conform to their public interface, allowing (in theory) non-bypassable security checks to be embedded in appropriate places.

Although memory safety via type safety is a well known concept, theory can be different than implementation. For example, it is widely believed that Standard ML is type safe. However, the SML / NJ compiler uses a 128-bit cyclic redundancy code (CRC) as the fingerprint of a `signature` [AM94]. Because CRCs are not cryptographically secure, it should be possible to generate two different signatures with the same fingerprint. The SML / NJ compiler will then generate unsound code. Of course, this is an implementation artifact of SML / NJ — for non-malicious applications, the 128-bit CRC is perfectly adequate. The random chance of a fingerprint collision is negligible, but an attacker can cause a collision to occur.

Such implementation artifacts point to the difficulty of producing systems that are actually type safe. More important, however, are higher level design problems. We have seen several failures of type and memory safety in Java related to the (mis-)design of dynamic linking. This thesis makes three contributions:

1. We examine the security of three popular Java implementations;
2. We give a model of dynamic linking in PVS and formally prove that it preserves type safety, assuming that Java without dynamic linking is type-safe;

3. We explore the use of binary decision diagrams (BDDs) for analyzing cryptographic hash functions, which are critical for supporting digitally signed mobile code.

The history of formal methods in the security arena is a long one. Much work was done in the late 1970s and early 1980s attempting to verify multi-level security properties of operating systems. PSOS [NBF+80] was one typical example. The Orange Book [Nat85] requires mechanized formal methods for high-assurance (i.e., A1-rated) systems. However, these techniques have not been widely used. Only one A1-rated general purpose operating system has been evaluated in 15 years. The principal problems have been the difficulty of the verification, and the specification-to-implementation correspondence problem. Advances in tools and techniques, along with vast computational speedups (2–3 orders of magnitude), have significantly eased the difficulty of verification. Model checking promises vastly increased automation for verification problems that can be reduced to a finite state description. The specification-to-implementation correspondence problem remains; however, modern programming languages (*e.g.*, Java, SML) should help there.

There is a long running debate in the formal methods community over the relative merits of theorem proving vs. model checking. We are agnostic about this debate. The relative difficulty and effectiveness of searching finite state spaces vs. generating (inductive) proofs are arguments best left to others. We attempt to use the best tools available for each problem. For modeling dynamic linking, we desire the power to model an unbounded system, which guides us toward PVS' higher-order logic. For the cryptographic hash functions, where we have a finite problem that is somewhat amenable to a heuristically guided brute-force approach, we use

BDDs, trading off expressive power for dramatically more automation. Because the problem domain is a combinational circuit, we do not need to use the model checker's temporal logic facilities, but BDD implementations are often embedded in model checkers, which we can use off the shelf.

Language I have spent enough time attacking these systems that I often use first person to describe the attacker. This also solves the problem of English not providing a gender-neutral singular pronoun; without prejudice, "his" is used throughout.

Chapter 2

Java Security[1]

2.1 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 that could lead to the execution of arbitrary machine code [Cro95]. When the Web left the research domain and entered the mass market, security became a problem: users wanted electronic commerce. The Secure Socket Layer (SSL) [FKK96, DA97, Gho98], and Secure HyperText Transport Protocol (S-HTTP) [SR98, Gho98] protocols were designed to provide cryptographically strong identification of Web servers, and privacy protection for information such as credit card numbers. Although an early implementation of SSL had a problem seeding its random number generator [GW96], and cryptographic protocols are always tricky to design, the situation appeared to be well in hand.

[1]Earlier versions of this chapter appeared in the *1996 IEEE Symposium on Security and Privacy* [DFW96] and *Internet Besieged: Countering the Cyberspace Scofflaw* [DFWB97].

Rather than creating new HTML extensions, Sun Microsystems popularized the notion of downloading a program (called an *applet*) that runs inside the Web browser. Such remote code raises serious security issues; a casual Web reader should not be concerned about malicious side-effects from visiting a Web page. Languages such as Java [GJS96], Safe-Tcl [Bor94], Phantom [Cou95], Juice [FK97] and Telescript [Gen96] have been proposed for running untrusted code, and each has varying ideas of how to thwart malicious programs. Java has dominated the market for Web-based mobile code. Java was promoted as addressing the security issue, in that browsers were supposed to be protected from unfriendly actions by the applets that they run; however, numerous problems have been found, as we will describe.

After several years of development inside Sun Microsystems, the Java language was released in mid-1995 as part of Sun's HotJava Web browser. Shortly thereafter, Netscape Communications Corp. announced they had licensed Java and would incorporate it into their Netscape Navigator Web browser, beginning with version 2.0. Microsoft has also licensed Java from Sun, and incorporated it into Microsoft Internet Explorer 3.0. With the support of many influential companies, Java appears to have the best chance of becoming the standard for mobile code on the Web. This also makes it an attractive target for malicious attackers, and demands external review of its security.

The original version of this chapter was written after Netscape announced it would use Java. Since that time, we have found a number of bugs in Navigator through its various beta and production releases and later in Microsoft's Internet Explorer. As a direct result of our investigation, and the tireless efforts of the vendors' Java programmers, we believe the security of Java has significantly improved since its early days. In particular, Internet Explorer 3.0, which shipped in August, 1996, had the benefit of nine months of our investigation into Sun's and Netscape's Java implementations. Still, despite all the work done by us and by others, Java's security problems are still fertile grounds for exploration.

Netscape Navigator and HotJava[2] are examples of two distinct architectures for building Web browsers. Netscape Navigator is written in an unsafe language, C, and runs Java applets as an add-on feature. HotJava is written in Java itself, with the same runtime system supporting both the browser and the applets. Both architectures have advantages and disadvantages with respect to security: Netscape Navigator can suffer from being implemented in an unsafe language (buffer overflow, memory leakage, etc.), but provides a well-defined interface to the Java subsystem. In Netscape Navigator, Java applets can name only those functions and variables explicitly exported to the Java subsystem. HotJava, implemented in a safe language, does not suffer from potential memory corruption problems, but can accidentally export private browser state to applets.

In order to be secure, such systems must limit applets' access to system resources such as the file system, the CPU, the network, the graphics display, and the browser's internal state. Additionally, the system should garbage-collect memory to prevent both malicious and accidental memory leakage. Finally, the system must manage system calls and other methods that allow applets to affect each other as well as the environment beyond the browser.

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

[2]Unless otherwise noted, in this chapter, "HotJava-Alpha" refers to the 1.0 alpha 3 release of the HotJava Web browser from Sun Microsystems, "Netscape Navigator" refers to Netscape Navigator 2.0, "Internet Explorer" refers to Microsoft Internet Explorer 3.0, and "JDK" refers to the Java Development Kit, a Java compiler and execution environment, version 1.0, from Sun.

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 will work as documented. We also require that the system be *memory safe*, such that a program can only access memory locations that hold values supposed to be accessible by that program. All security in Java depends upon these properties being enforced. If type safety fails, then a Java program can access `private` fields of objects, and cause behavior not envisioned the original programmer. Some type safety failures allow integers to be used as pointers, allowing nearly arbitrary access to the underlying machine, at least within the confines of the current process in the underlying operating system (if any). “Arbitrary access to the underlying machine” implies that Java does not supply any security beyond the underlying operating system: if the operating system would allow the user running the Java program to, *e.g.*, erase a file, then the Java program could also. This is clearly an unacceptable state of affairs for running untrusted mobile code. Although 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.

Many systems in the past have attempted to use language-based protection. The Burroughs B5000 series appears to have been the first notable attempt; its descendents live on today as the Unisys A-series [Uni94]. The Anderson report [And72] describes an early attempt to build a secure subset of Fortran. This effort was a failure because the implementors failed to consider all of the consequences of the implementation of one construct: assigned `GOTO`. This subtle flaw resulted in a complete break of the system. Jones and Liskov describe language support for secure data flow [JL76]. The Pilot operating system as Xerox PARC used type safety for memory protection, but it attempted only to prevent mistakes from crashing the machine; it did not attempt to thwart malice. For a single-user workstation, occasionally rebooting was considered acceptable [R+80]. Rees describes a modern capability system built on top of Scheme [Ree96].

Vigna recently edited a nice collection of papers [Vig98]. Chess [Che98] provides an overview of the problems in mobile code security. Volpano and Smith [VS98] examine information flow in language based security. Other relevant papers in this collection include works by Necula and Lee [NL98], Gray, *et al.*, [GKCR98], Karjoth, *et al.*, [KLO98], and Gong and Schemers [GS98].

The remainder of this chapter is structured as follows. Section 2.2 discusses the Java language in more detail, Section 2.3 discusses the original implementation of Java security mechanisms, Section 2.4 gives a taxonomy of known security flaws in Sun's HotJava, Netscape's Navigator, and Microsoft's Internet Explorer Web browsers, Section 2.6.4 discusses the need for accountability in Java, and Section 2.7 presents our conclusions. A slower paced discussion, aimed at a less technical audience, of some of these issues can be found in McGraw and Felten's book [MF96].

2.2 Java Semantics

Java is similar in many ways to C++ [Str91]. Both provide support for object-oriented programming, share many keywords and other syntactic elements, and can be used to develop standalone applications. Java diverges from C++ in the following ways: it is type-safe, supports only single inheritance (although it decouples subtyping from inheritance), and has language support for concurrency. Java supplies each class and object with a lock, and provides the `synchronized` keyword so each class (or instance of a class, as appropriate) can operate as a Mesa-style monitor [LR80].

Java compilers produce a machine-independent bytecode, which may be transmitted across a network and then interpreted or compiled to native code by the Java runtime system. In support of this downloaded code, Java distinguishes *remote* code from *local* code. Separate sources[3] of Java bytecode are loaded in separate name spaces to prevent both accidental and malicious name clashes. Bytecode loaded from the local file system is visible to all applets. The documentation [GM96] says the "system name space" has two special properties:

1. It is shared by all "name spaces."

2. It is always searched first, to prevent downloaded code from overriding a system class; see Section 2.4.5.

However, we have found that the second property does not hold.

The Java runtime system knows how to load bytecode only from the local file system; it has no built in knowledge of the Internet. To load code from other sources, the Java runtime system calls a subclass of the `abstract` class[4] `ClassLoader`, which defines an interface for the runtime system to ask a Java program to provide a class. Classes are transported across the network as byte streams, and reconstituted into `Class` objects by subclasses of `ClassLoader`. Each class is internally tagged with the `ClassLoader` that loaded it, and that `ClassLoader` is used to resolve any future unresolved symbols for the class. Additionally, the `SecurityManager` has methods to determine if a class loaded by a

[3]Although the documentation [GM96] does not define "source", it appears to mean the URL prefix of origin. Sun, Netscape, and Microsoft all offer support for digitally signed code. See Dan Wallach's thesis [Wal99] for more details.

[4]An `abstract` class is a class with one or more methods declared but not implemented. Abstract classes cannot be instantiated, but define method signatures for subclasses to implement.

ClassLoader is in the dynamic call chain, and if so, where. This nesting depth is then used to make access control decisions in JDK 1.0.*x* and derived systems (including Netscape Navigator 2.0.*x* and Internet Explorer 3.0).

Java programmers can combine related classes into a package. These packages are similar to name spaces in C++ [Str94], modules in Modula-2 [Wir83], or structures in Standard ML [MTHM97]. Although package names consist of components separated by dots, the package name space is actually flat: scoping rules are not related to the apparent name hierarchy. In Java, public and private have the same meaning as in C++: Public classes, methods, and instance variables are accessible everywhere, whereas private methods and instance variables are accessible only inside the class definition. Java protected methods and variables are accessible in the class or its subclasses or in the current (package, ClassLoader) pair. A (package, ClassLoader) pair defines the scope of a Java class, method, or instance variable that is not given a public, private, or protected modifier[5]. Unlike C++, protected variables and methods can be accessed in subclasses only when they occur in instances of the subclasses or further subclasses. For example:

```
class Foo {
  protected int x;
  void SetFoo(Foo obj) { obj.x = 1; } // Legal
  void SetBar(Bar obj) { obj.x = 1; } // Legal
}


class Bar extends Foo {
  void SetFoo(Foo obj) { obj.x = 1; } // Illegal
  void SetBar(Bar obj) { obj.x = 1; } // Legal
}
```

[5]Methods or variables with no access modifiers are said to have *package scope*.

The definition of `protected` was the same as C++ in some early versions of Java; it was changed during the beta-test period to patch a security problem [Mue96] (see also section 2.6.2). The present definition, while complicated, seems to offer some software engineering benefits: classes cannot change the state of `protected` variables of instances of parent or sibling classes. This should increase the modularity of code in general.

The Java Virtual Machine (JVM) is designed to enforce the language's access semantics. Unlike C++, programs are not permitted to forge a pointer to a function and invoke it directly, nor to forge a pointer to data and access it directly. If a rogue applet attempts to call a private method, the runtime system throws an exception, preventing the errant access. Thus, if the system libraries are specified safely, the runtime system is designed to ensure that application code cannot break these specifications.

The Java documentation claims that the safety of Java bytecodes can be statically determined at load time. This is not entirely true: the type system uses a covariant [Cas95] rule for subtyping arrays, so array stores require run time type checks[6] in addition to the normal array bounds checks. Cast expressions also require runtime checks. Unfortunately, this means the bytecode verifier is not the only piece of the runtime system that must be correct to ensure type safety; the interpreter must also perform runtime checks with a corresponding performance degradation.

[6]For example, suppose that `A` is a subtype of `B`; then the Java typing rules say that `A[]` ("array of `A`") is a subtype of `B[]`. Now the following procedure cannot be statically type-checked:

```
void proc(B[] x, B y) {
  x[0] = y;
}
```

Because `A[]` is a subtype of `B[]`, `x` could really have type `A[]`; similarly, `y` could really have type `A`. The body of `proc` is not type-safe if the value of `x` passed in by the caller has type `A[]` and the value of `y` passed in by the caller has type `B`. This condition cannot be checked statically.

2.3 Java Security Mechanisms

In HotJava-Alpha, the access controls were done on an ad hoc basis that was clearly insufficient. The beta release of JDK introduced the `SecurityManager` class, meant to be a reference monitor [Lam71]. The `SecurityManager` implements a security policy, centralizing all access control decisions. All potentially dangerous methods first consult the security manager before executing. Netscape and Microsoft also used this architecture through the version 3.*x* releases of their products.

When the Java runtime system starts up, there is no security manager installed. Before executing untrusted code, it is the Web browser's or other user agent's responsibility to install a security manager. The `SecurityManager` class is meant to define an interface for access control; the default `SecurityManager` implementation throws a `SecurityException` for all access checks, forcing the user agent to define and implement its own policy in a subclass of `SecurityManager`. The security managers in current browsers typically make their access control decisions by examining the contents of the call stack, looking for the presence of a `ClassLoader`, indicating that they were called, directly or indirectly, from an applet.

Java uses its type system to provide protection for the security manager. If Java's type system is sound, then the security manager, in theory, will be tamperproof. By using types instead of separate address spaces for protection, Java is more easily embeddable in other software, and potentially performs better because protection boundaries can be crossed without a context switch [ALBL91]. Interfaces can be more expressive than integers and strings — objects obeying invariants (on their private state) can be created by trusted code and given to untrusted code, knowing that those invariants will be preserved. This should simplify the development of robust software.

2.4 Taxonomy

We now present a taxonomy of known Java bugs, past and present. Dividing the bugs into categories is useful because it helps us understand how and why they arose, and it alerts us to aspects of the system that may harbor future bugs.

2.4.1 Denial-of-Service Attacks

Java has few provisions to thwart denial-of-service attacks. Obvious attacks are busy-waiting to consume CPU cycles and allocating memory until the system runs out, starving other threads and system processes. Additionally, an applet can acquire locks on critical pieces of the browser to cripple it. *E.g.*, the code in Figure 2.1 locks the status line at the bottom of the HotJava-Alpha browser, effectively preventing it from loading any more pages. In Netscape Navigator 2.0, this attack can lock the `java.net.InetAddress` class, blocking all hostname lookups and hence most new network connections. HotJava, Navigator, and Internet Explorer all have classes suitable for this attack. The attack could be prevented by replacing such critical classes with wrappers that do not expose the locks to untrusted code. However, the CPU and memory attacks cannot be easily fixed; many genuine applications need large amounts of memory and CPU. Another attack, first implemented by Mark LaDue, is to open a large number of windows on the screen. This

```
synchronized (Class.forName("net.www.html.MeteredStream")) {
    while(true) Thread.sleep(10000);
}
```

Figure 2.1: Java code fragment to deadlock the HotJava browser by locking its status line.

will sometimes crash the machine. LaDue has a Web page with many other denial-of-service attacks [LaD].

There are two twists that can make denial-of-service attacks more difficult to cope with. First, an attack can be programmed to occur after some time delay, causing the failure to occur when the user is viewing a different Web page, thereby masking the source of the attack. Second, an attack can cause *degradation of service* rather than outright denial of service. Degradation of service means significantly reducing the performance of the browser without stopping it. For example, the locking-based attack could be used to hold a critical system lock most of the time, releasing it only briefly and occasionally. The result would be a browser that runs very slowly.

Sun has said that they consider denial-of-service attacks to be low-priority problems [Gos95]. Given the other major problems, this seems to be a reasonable position. However, anyone depending on timely execution of an applet (*e.g.*, for commodity trading) may be disappointed.

2.4.2 Confidentiality Breaches

Protecting confidentiality of users' data is a key goal of the Java security policy in Web browsers. There are two ways of achieving this goal: preventing applets from learning about the environment they are running in, and blocking the channels to communicate this information to the outside world. The default applet security policy only allows an applet to make a network connection to the host it was loaded from. This policy is a compromise between security and utility. Communication with other hosts is prohibited by policy, but we show that this policy can be subverted via the use of covert channels.

Information Available to Applets

In HotJava-Alpha, most attempts by an applet to read or write the local file system result in a dialog box for the user to grant approval. Separate access control lists (ACLs)[7] specify where reading and writing of files or directories may occur without the user's explicit permission. By default, the write ACL is empty and the read ACL contains the HotJava library directory and specific MIME `mailcap` files. The read ACL also contains the user's `public_html` directory, which may contain information which compromises the privacy of the user. The Windows 95 version additionally allows writing (but not reading) in the `\TEMP` directory. This allows an applet to corrupt files in use by other Windows applications if the applet knows or can guess names the files may have. At a minimum, an applet can consume all the free space in the file system. These security concerns could be addressed by the user editing the ACLs; however, the system default should have been less permissive. Navigator and Internet Explorer do not permit any file system access by applets (without digital signatures).

In HotJava-Alpha, we could learn the user's login name, machine name, as well as the contents of all environment variables; `System.getenv()` in HotJava-Alpha had no security checks. By probing environment variables, including the `PATH` variable, we could often discover what software is installed on the user's machine. This information could be valuable either to corporate marketing departments, or to attackers desiring to break into a user's machine. In later Java versions, `System.getenv()` was replaced with "system properties," many of which are not supposed to be accessible by applets. However, there have been implementation problems (see Section 2.4.5) that allowed an applet to read or write any

[7]Although Sun calls these "ACLs", they are actually *profiles* — a list of files and directories granted specific access permissions.

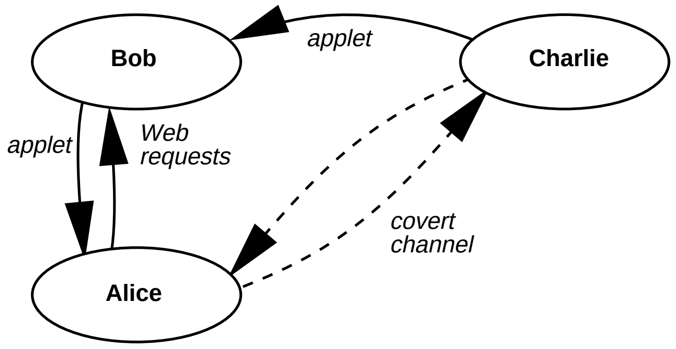

Figure 2.2: A Three Party Attack — Charlie produces a Trojan horse applet. Bob likes it and uses it in his Web page. Alice views Bob's Web page and Charlie's applet establishes a covert channel to Charlie. The applet leaks Alice's information to Charlie. No collusion with Bob is necessary.

system property.

Java allows applets to read the system clock, making it possible to benchmark the user's machine. As a Java-enabled Web browser may well run on pre-release hardware and/or software, an attacker could learn valuable information. Timing information is also needed for the exploitation of covert timing channels. "Fuzzy time" [Hu91] should be investigated to see if it can mitigate these problems.

Two vs. Three Party Attacks

It is useful to distinguish between two different kinds of attack, which we shall call *two-party* and *three-party*. A two-party attack requires that the Web server the applet resides on participate in the attack. A three-party attack can originate from anywhere on the Internet, and might spread if it is hidden in a useful applet that gets used by many Web pages (see Figure 2.2). Three-party attacks are more dangerous than two-party attacks because they do not require the collusion of the Web server.

Covert Channels

Various covert channels exist in HotJava, Navigator, and Internet Explorer, allowing applets to have two-way communication with arbitrary third parties on the Internet. These are not classic timing or storage channels, but exploit mechanisms not intended to be used for data communication.

Typically, most HotJava users will use the default network security mode, which allows an applet to connect only to the host from which it was loaded. This is the only security mode available to Navigator and Internet Explorer users[8]. In fact, the browsers have failed to enforce this policy through a number of errors in their implementation.

The `accept()` system call, used to receive a network connection initiated on another host, is not protected by the usual security checks in HotJava-Alpha. This allows an arbitrary host on the Internet to connect to a HotJava browser as long as the location of the browser is known. For this to be a useful attack, the applet needs to signal the external agent to connect to a specified port. Even an extremely low-bandwidth covert channel would be sufficient to communicate this information. The `accept()` call is properly protected in current Java implementations.

If the Web server that provided the applet is running an SMTP mail daemon, the applet can connect to it and transmit an e-mail message to any machine on the Internet. Additionally, the *Domain Name System* (DNS) can be used as a two-way communication channel to an arbitrary host on the Internet. An applet may reference a fictitious name in the attacker's domain. This transmits the name to the attacker's DNS server, which could interpret the name as a message, and then send a list of arbitrary 32-bit IP numbers as a reply. Repeated DNS calls by the applet

[8]Without using digitally signed code.

establish a channel between the applet and the attacker's DNS server. This channel also passes through a number of firewalls [CB94]. In HotJava-Alpha, the DNS channel was available even with the security mode set to "no network access," although this was fixed in later Java versions. DNS has other security implications; see section 2.4.3 for details.

Another third-party channel is available with the *URL redirect* feature. Normally, an applet may instruct the browser to load any page on the Web. An attacker's server could record the URL as a message, then redirect the browser to the original destination.

When we notified Sun about these channels, they said the DNS channel would be fixed [Mue95], but in fact it was still available in JDK 1.0 and Netscape Navigator 2.0. Netscape has since issued a patch (incorporated into Netscape Navigator 2.01) to fix this problem.

As far as we know, nobody has done an analysis of covert storage or timing channels in the Java runtime system for evaluating isolation between applets.

2.4.3 Implementation Errors

Some bugs arise from fairly localized errors in the implementation of the browser or the Java subsystem.

DNS Weaknesses

A significant problem appeared in the JDK 1.0 and Netscape Navigator 2.0 implementation of the policy that an applet can open a TCP/IP connection only back to the server it was loaded from. While this policy is reasonable, since applets often need to load components (images, sounds, etc.) from their host, it was not uniformly enforced. This policy was enforced as follows:

1. Get all the IP addresses of the hostname that the applet came from.
2. Get all the IP addresses of the hostname that the applet is attempting to connect to.
3. If any address in the first set matches any address in the second set, allow the connection. Otherwise, do not allow the connection.

The problem occurred in the second step: the applet can ask to connect to any hostname on the Internet, so it can control which DNS server supplies the second list of IP-addresses; information from this untrusted DNS server was used to make an access control decision. There is nothing to prevent an attacker from creating a DNS server that lies [Bel95]. In particular, it may claim that any name for which it is responsible has any given set of addresses. Using the attacker's DNS server to provide a pair of addresses *(machine-to-connect-to, machine-applet-came-from),* the applet could connect to any desired machine on the Internet. The applet could even encode the desired IP-address pair into the hostname that it looks up. This attack is particularly dangerous when the browser is running behind a firewall, because the malicious applet can attack any machine behind the firewall. At this point, a rogue applet can exploit a whole legion of known network security problems to break into other nearby machines.

This problem was postulated independently by Steve Gibbons [Gib96] and by us. To demonstrate this flaw, we produced an applet that exploits an old `sendmail` hole to run arbitrary Unix commands as user `daemon`.

```
hotjava.props.put("proxyHost", "proxy.attacker.com");
hotjava.props.put("proxyPort", "8080");
hotjava.props.put("proxySet", "true");
HttpClient.cachingProxyHost = "proxy.attacker.com";
HttpClient.cachingProxyPort = 8080;
HttpClient.useProxyForCaching = true;
```

Figure 2.3: Code to redirect all HotJava-Alpha HTTP retrievals. FTP retrievals may be redirected with similar code.

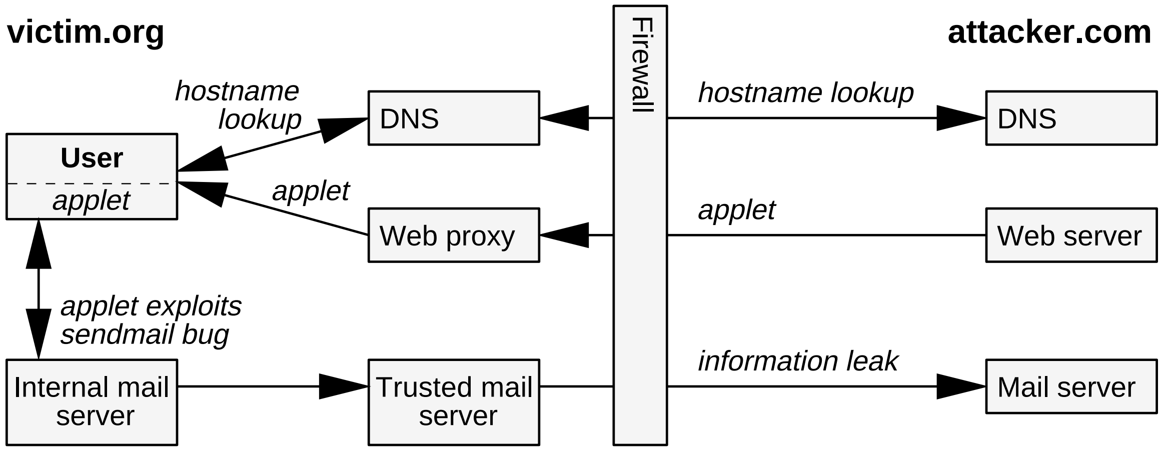

Figure 2.4: DNS subversion of Java: an applet travels from `attacker.com` to `victim.org` through normal channels. The applet then asks to connect to `foo.attacker.com`, which is resolved by the DNS server for `attacker.com` to be mail server inside `victim.org` which can then be attacked.

Sun (JDK 1.0.1) and Netscape (Navigator 2.01)[9] have both issued patches to fix this problem.

[9]Netscape solved the problem by storing the results of all DNS name lookups internally, forcing a given hostname to map to exactly one IP address. Netscape Navigator also stores the applet source as a function of its IP address, not hostname. This solution has the added property that it prevents time-varying DNS attacks. Previously, an attacker's name server could have returned different IP addresses for the same hostname each time it was queried, allowing the same attacks detailed above.

Buffer Overflows

HotJava-Alpha had many unchecked `sprintf()` calls that used stack-allocated buffers. Because `sprintf()` does not check for buffer overflows, an attacker could overwrite the execution stack, thereby transferring control to arbitrary code. Attackers have exploited the same bug in the Unix `syslog()` library routine (via `sendmail`) to take over machines from across the network [CER95]. In later Java releases, all of these calls were fixed in the Java runtime. However, the bytecode disassembler was overlooked all the way through the JDK 1.0 release. Users disassembling Java bytecode using **javap** were at risk of having their machines compromised if the bytecode had very long method names. This bug was fixed in JDK 1.0.2.

Disclosing Storage Layout

Although the Java language does not allow direct access to memory through pointers, the Java library allows an applet to learn where in memory its objects are stored. All Java objects have a `hashCode()` method which, in typical implementation (including early Sun and Netscape implementations) unless overridden by the programmer, casts the address of the object's internal storage to an integer and returns it. Although this does not directly lead to a security breach, it exposes more internal state than necessary.

Public Proxy Variables

An interesting attack on HotJava-Alpha is that an attacker can change the browser's HTTP and FTP proxy servers. An attacker can establish his own proxy server as a man-in-the-middle. As long as the client is using unencrypted HTTP and FTP protocols, we can both watch and edit all traffic to and from the HotJava-Alpha browser. All this is possible simply because the browser state was stored in public variables in public classes. This attack compromises the user's privacy, and its implementation is trivial (see Figure 2.3). By using the property manager's `put()` method, an attackers stores a desired proxy in the property manager's database. If the attacker can then entice the user to print a Web page, these settings will be saved to disk, and will be the default settings the next time the user starts HotJava. If the variables and classes were private, this attack would fail. Likewise, if the browser were running behind a firewall and relied on proxy servers to access the Web, this attack would also fail.

We note that the same variables are `public` in JDK 1.0, although they are not used. This code is not part of Navigator or Internet Explorer.

2.4.4 Interapplet Security

Because applets can persist after the Web browser leaves the page that contains them, it becomes important to protect applets from each other. Otherwise, an attacker's applet could deliberately sabotage a third party's applet. In many environments, it would be unacceptable for an applet to even learn of the existence of another applet. Non-interference [GM82, GM84] is a formal definition that captures this intuition.

In Netscape Navigator 3.0, `AppletContext.getApplets()` is careful to return handles only to applets on the same Web page as the caller. However, an applet could easily get a handle to the top-level `ThreadGroup` and then enumerate every thread running in the system, including threads belonging to other arbitrary applets. The Java runtime encodes the applet's class name in its thread name, so a rogue applet can now learn the names of all applets running in the system. In addition, an applet could call the `stop()` or `setPriority()` methods on threads in other applets. The `SecurityManager` checked only that applets could not alter the state of system threads; there were no restraints on applets altering other applet threads. Netscape Navigator 4.0 prevents an attacker from seeing threads belonging to applets on other Web pages, in the same way it protects applets. Internet Explorer allows an applet to see those threads, but calls to `stop()` or `setPriority()` have no effect.

An insidious form of this attack involves a malicious applet that lies dormant except when a particular victim applet is resident. When the victim applet is running, the malicious applet randomly mixes degradation of service attacks with attacks on the victim applet's threads. The result is that the user sees the victim applet as slow and buggy.

2.4.5 Java Language Implementation Failures

Unfortunately, the Java language and the bytecode it compiles to are not as secure as they could be. There are significant differences between the semantics of the Java language and the semantics of the bytecode language. First, we discuss David Hopwood's attack [Hop96b] based on package names. Next, we present our attack that runs arbitrary machine code after compromising the type system. Several flaws in the type system are examined, including two first noted by Tom Cargill.

Illegal Package Names

Java packages are normally named `java.io`, `java.net`, etc. The language prohibits "." from being the first character in a package name. The runtime system replaces each "." with a "/" to map the package hierarchy onto the file system hierarchy; the compiled code is stored with the periods replaced with slashes. David Hopwood found that if the first character of a package name was "/", the Java runtime system would attempt to load code from an absolute path [Hop96b], because absolute pathnames begin with a "/" character on Unix. Thus, if an attacker could place compiled Java in any file on the victim's system (either through a shared file system, via an incoming FTP directory, or via a distributed file system such as AFS), the attacker's code would be treated as trusted, because it came from the local file system rather than from the network. Trusted code is permitted to load dynamically linked libraries (DLLs, written in C) which can then ignore the Java runtime and directly access the operating system with the full privileges of the user.

This attack is actually more dangerous than Hopwood first realized. Since Netscape Navigator caches the data it reads in the local file system, Netscape Navigator's cache can also be used as a way to get a file into the local file system. In this scenario, a normal Java applet would read (as data) files containing bytecode and DLL code from the server where the applet originated. The Java runtime would ask Navigator to retrieve the files; Navigator would deposit them in the local cache. As long as the applet can figure out the file names used by Navigator in its cache, it can execute arbitrary machine code without even needing prior access to the victim's file system.

JDK 1.1 Signature Management

In the JDK 1.1.1 release from Sun, there was a problem in the management of digitally-signed code. The model in JDK 1.1 is that signed code, if the signature is accepted by the user, receives the ability to do anything. Unfortunately, there were two problems:

1. The method `java.security.IdentityScope.identities` would enumerate all the signers known to the system. This is bad design.

2. The method `java.lang.Class.getSigners` returned a reference to the array of identity objects used by the runtime to track the signer of the current class. This array is writable. The code should have returned a copy of the array instead, so that modifications would not be used by the JVM.

The net result of these problems was to make it a trivial exercise to write an applet which gets access to all the signers that a JVM knows about, and tries on identities, one at a time, until it finds a privileged one. Thus, if any signer is trusted, all applets are capable using the privileges of the trusted signer. This problem was fixed in JDK 1.1.2.

Superclass Constructors

The Java language [GJS96] requires that all constructors call either another constructor of the same class or a superclass constructor as their first action. The system classes `ClassLoader`, `SecurityManager`, and `FileInputStream` all rely on this behavior for their security. These classes have constructors that check if they are called from an applet, and throw a `SecurityException` if so. Unfortunately, although the Java language prohibits the following code, the bytecode verifier readily accepted its bytecode equivalent:

```
class CL extends ClassLoader {
    CL() {
        try { super(); }
        catch (Exception e) {}
    }
}
```

This allowed an attacker to build (partially uninitialized) `ClassLoaders`, `SecurityManagers`, and `FileInputStreams`. `ClassLoaders` are the most interesting class to instantiate, as any code loaded by a `ClassLoader` asks its `ClassLoader` to resolve any classes it references. This is contrary to the documentation [GM96] that claims the system name space is always searched first; we have verified this difference experimentally. Fortunately for an attacker, `ClassLoaders` did not have any instance variables, and the `ClassLoader` constructor needs to run only once, to initialize a variable in the runtime system. This happens before any applets are loaded. Therefore, this attack resulted in a properly initialized `ClassLoader` which is under the control of an applet. Since `ClassLoaders` define the name space seen by other Java classes, the applet can construct a completely customized name space. A fix for this problem appeared in Netscape Navigator 2.02, which was later broken (see Section 2.4.5). Netscape Navigator 3.0 and JDK 1.0.2 took different approaches to fix this problem.

We discovered that creating a `ClassLoader` gives an attacker the ability to defeat Java's type system. Assume that classes A and B both refer to a class named C. A `ClassLoader` could resolve A against class C, and B against class C'. If an object of class C is allocated in A, and then is passed as an argument to a method of B, the method in B will treat the object as having a different type, C'. If the fields of C' have different types (e.g., `Object` and `int`) or different access modifiers (`public, private, protected`) than those of C, then Java's type safety is defeated.

This allows an attacker to get and set the value of *any* non-static variable, and call *any* method (including native methods). This attack also allows an applet to modify the class hierarchy, as it can read and write variables normally visible only by the runtime system. Any attack which allows object references to be used as integers, and *vice versa*, leads to complete penetration of Java (see section 2.4.5). Java's bytecode verification and class resolution mechanisms are unable to detect these inconsistencies because Java defines a weak correspondence between class names and Class objects.

Netscape Navigator 3.0 and Microsoft Internet Explorer fix the superclass constructor issue and take other measures to prevent applets from instantiating ClassLoaders. JDK 1.1-Beta initially offered "safe" ClassLoaders to applets, but the feature was withdrawn from the final release because they could, in fact, still be abused.

Fundamentally, the job of a ClassLoader is to resolve names to classes as part of Java's dynamic linking. Dynamic linking has subtle interactions with static type checking. This problem has driven much of my research; Chapters 3 and 4 are devoted to it.

Attacking the SecurityManager

Unfortunately, a ClassLoader can load a new SecurityManager that redeclares the SecurityManager's variables as public, violating the requirement that reference monitors be tamperproof. There are four interesting variables in the JDK AppletSecurity class: readACL, writeACL, initACL, and networkMode. The readACL and writeACL variables are lists of directories and files that applets are allowed to read and write. The initACL variable tracks whether the ACLs have been initialized. The networkMode variable determines what hosts applets are allowed to make network connections to. By setting the networkMode variable to allow connections anywhere, the ACLs to null, and the initACL variable to true, we effectively circumvent all JDK security.

Java's SecurityManagers generally base their access control decisions on whether they are being called in the dynamic scope of code loaded from the network. The default ClassLoader in the runtime system knows how to load classes only from the local file system, and appears as the special value NULL to the runtime system. Any other ClassLoader is assumed to indicate untrusted code. However, a ClassLoader can provide an implementation of Class that makes certain runtime system data structures accessible as ints. Setting the ClassLoader field to zero causes the Java runtime system to believe that the code came from the local file system, also effectively bypassing the SecurityManager.

Running Machine Code from Java

Netscape Navigator 2.0*x* protected itself from the attacks described above with additional checks in the native methods that implement file system routines that applets would never have any reason to invoke. In addition, the java.lang.Runtime.exec() method has been stubbed out in the runtime system, so applets cannot start Unix processes running. However, the type system violations (i.e., using Objects as ints and *vice versa*) make it possible, but non-trivial, to run arbitrary machine code, at which point an attacker can invoke any system call available to the user running the browser without restriction, and thus has completely penetrated all security provided by Java.

Although Java does not guarantee a memory layout for objects [LY96], the current implementations lay out objects in the obvious way: instance variables are in consecutive memory addresses, and packed as in C. An attacker can clearly write machine code into integer fields, but there are two remaining challenges: learning the memory address of our code, and arranging for the system to invoke our machine code. All an attacker can do is use object references as integers, but note that object references in JDK and Netscape Navigator are pointers to pointers to objects, and we can only doubly dereference them; thus, we cannot easily figure out where to jump to.

Internet Explorer uses a single pointer for object references, which simplifies the attack. Below, we describe how Netscape Navigator can be attacked, but this attack has been generalized to work with Microsoft Internet Explorer as well.

Netscape Navigator's object references are pointers to a structure that contains two pointers: one to the object's private data, and another to its type information. Thus, while a malicious `ClassLoader` allows an attacker to cast object references to integers and back, the attacker can only doubly dereference the pointers. This complicated the process of learning the actual machine address of an object's data.

To solve this problem, observe that `Class` objects, *i.e.*, instances of the class `java.lang.Class` are not implemented as normal Java objects. `Class` has no Java-visible variables, but the implementation of a `Class` by the runtime system uses a C structure, `ClassClass`, to hold class-specific data. The `ClassClass`, implemented in C, uses a mixture of Java object references and direct pointers. One of the fields of the `ClassClass` structure (which we have a reference to) contains a pointer to a method table. The method table contains a pointer back to the `ClassClass` and an array of pointers to method blocks; see Figure 2.5. Note that methods from superclasses are copied into each method table, and class `Object` has eleven methods, so

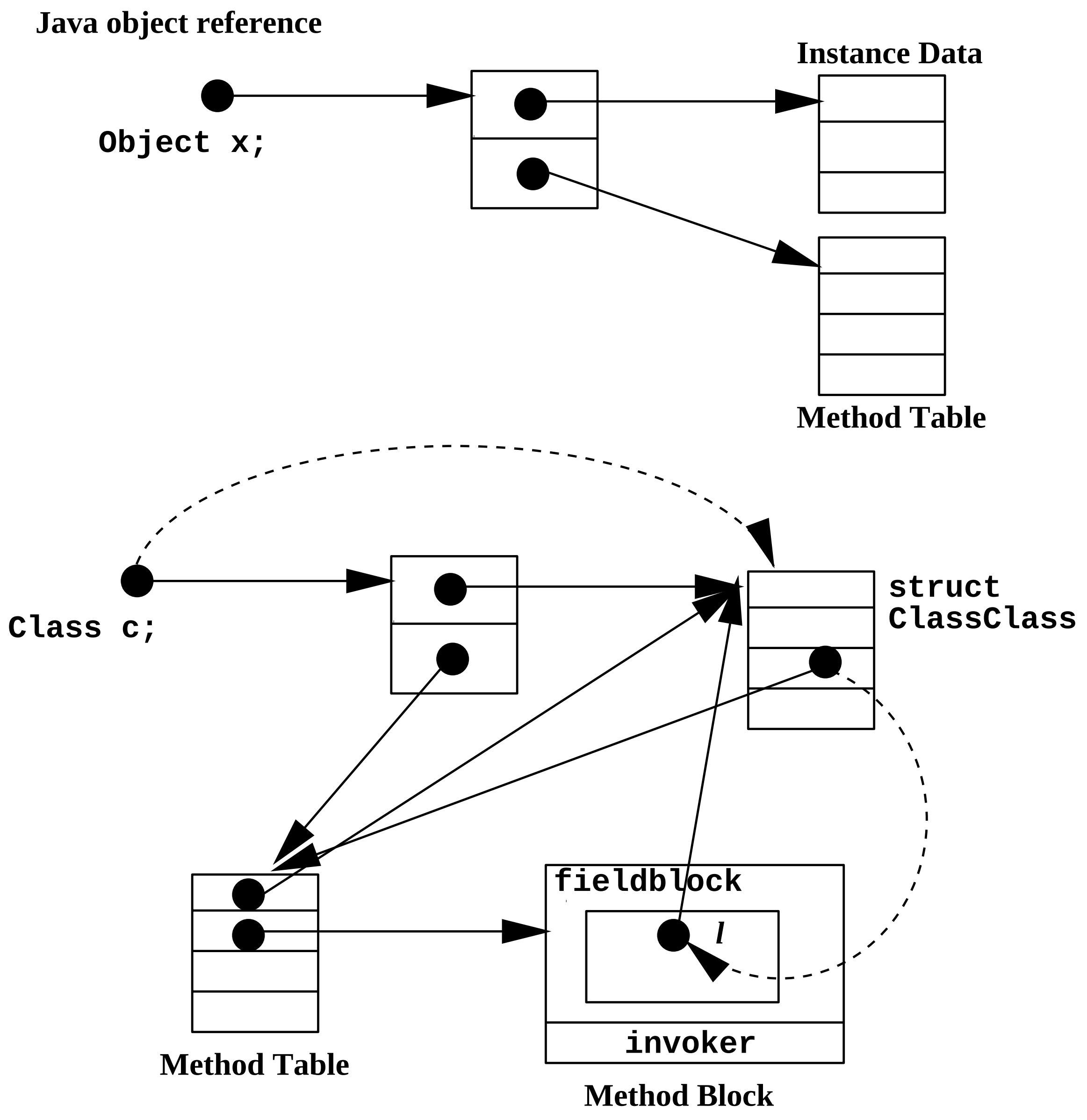

Figure 2.5: Memory layout of Java objects, and the special case of `java.lang.Class`.

```
lui    a0, ((a+20) div 65536)      ; Hi order bits of string pointer
addi   a0, a0, ((a+20) mod 65536)  ; Low order bits of string pointer
li     v0, 1010                    ; System call # for unlink() (IRIX)
syscall
nop
.asciz "/tmp/JavaSafe.NOT"         ; File to delete
```

Figure 2.6: MIPS assembly code to delete a file.

the twelfth method is the first method declared in this object. Each method block contains a pointer to its `ClassClass`.[10] Thus we have a cycle of 3 pointers, starting from the `Class` reference. Since 2 and 3 are relatively prime, we can cycle around the loop, gaining access to each and every pointer, in particular, getting access to a pointer to the `ClassClass`, which we treat as an integer *l*. We then overwrite memory beginning at *l* with the MIPS instructions[11] in Figure 2.6.

At this point, the machine code is in memory, and we know where it is. However, we've damaged a `ClassClass` structure, overwriting many of its fields. In particular, we've overwritten its constant pool pointer, so that methods can no longer be resolved by name. However, the Java interpreter resolves the method name the first time it is invoked, and replaces the method invocation with a so-called *quick* instruction, which caches the result of the lookup. By calling the method we hijack first, we cause the constant pool to be accessed, the method lookup result to be cached, and future calls to be performed with a quick-variant of the instruction. Each method block contains a function pointer to a function to be used for invoking that method: the runtime system contains functions for in-

[10]This pointer is actually the first field of the `fieldblock` structure, which in turn is the first field of the method block. However, the pointer to the `ClassClass` is at offset 0 from the beginning of the method block.

[11]We used an SGI workstation running version 5 of the IRIX operating system for the attack, because we wanted a 32-bit, big-endian architecture with simple calling conventions to simplify our job. The attack is equally applicable to other architectures and operating systems, but more complicated.

voking virtual methods, non-virtual methods, `synchronized` methods, and native methods. However, we can treat this function pointer as an integer, so we simply assign *l* (the beginning of the machine code) to it, and invoke the appropriate Java method. This, in turn, invokes our machine code. Our machine code will die with an illegal instruction as it tries to execute "/tmp", but that's fine — the damage has already been done.

More Type System Attacks

The `ClassLoader` attack described above was the first of many holes found in the Java type system. Any hole in the type system that allows integers to be used as object references is sufficient to run arbitrary machine code. Many such problems have been found.

Cargill's Interface Attack Tom Cargill noted that Java's `interface` feature could be used to call `private` methods [Car96a]. This works because all methods declared in an `interface` are `public`, and a class was allowed to implement an interface by inheriting a `private` method from its parent. Netscape Navigator 2.02 fixed the `ClassLoader` attack by making the native methods `private`, and wrapping them inside methods that checked a boolean variable initialized by the `ClassLoader` constructor before calling the `private, native` methods to do the real work. Because an attacker could now call the `private` methods directly, the fix was defeated.

Hopwood's Interface Attack David Hopwood found that `interfaces` were matched by string name, and not `Class` object identity [Hop96a]. By passing objects implementing an interface across multiple name spaces (where a name space corresponds to a ClassLoader), Hopwood could also treat object references as integers, and vice versa. Hopwood implemented his attack by storing a user-defined subclass of java.io.PrintStream in System.out, a public, non-final variable. Dirk Balfanz found that exceptions were also identified by name, and thus had the same problem [Bal96].

Arrays Java defines all arrays to be objects. The system normally provides array classes that inherit directly from Object, but use the covariant subtyping rule. However, we found that the user was able to define his own array classes because of a bug in the AppletClassLoader. When a class is loaded via Class.forName(), it will ask the ClassLoader of the code that invoked Class.forName() to supply the class, unless it's an array class, in which case the system will supply an appropriate definition. However, AppletClassLoader did not check that the name of the class it actually loaded is the same as what was requested until after it has called defineClass(), which had the side-effect of entering the class into the system class table. By misnaming an array class, *e.g.*, [LFoo; (*i.e.*, array of Foo), as a normal class, *e.g.*, Foo2, it was entered into the system class table, and could be used as an array. By calling Class.newInstance(), an attacker could allocate an instance of his class, and cast it into an array. When the class definition is needed to check the cast, the system first looks in the system class table, but only for array classes. If our definition has an integer as its first instance variable, and the array is Class[], then necessary conditions exist to run arbitrary machine code.

Tom Cargill noted that the *quick* variants of the method invocation instructions (which do not perform type checking) do not interact properly with arrays of interfaces [Car96b]. Recall that the method invocation instruction is rewritten at runtime, using the actual type of its argument the first time it is executed. However, not all elements of the array need have the same type. Using this, an attacker can call private methods of `public`, non-`final` classes. This was fixed in JDK 1.1, Navigator 3.0, and Internet Explorer 3.0.

Packages Dirk Balfanz found that the pre-release versions of Internet Explorer 3.0 did not separate packages with the same name loaded from different origins [BF96]. An applet could declare classes belonging to system packages such as `java.lang`. These classes would be accepted due to a error in the `SecurityManager`'s method that was supposed to prevent this. This error was also present in JDK 1.0.2. As a result, an applet could access package-scope variables [12] and methods of system packages. This bug was fixed in the final release of Internet Explorer 3.0. Because Sun and Netscape's implementation of the JVM detected this elsewhere, and did not in fact consult the `SecurityManager`, HotJava and Navigator were not vulnerable to this attack.

Exceptions In June 1998, we found another type soundness failure in the Netscape 4.05 JVM. The attack relies on several implementation problems in the interaction of `ClassLoaders` and exception handling. Mark LaDue found that one could instantiate a new `AppletClassLoader` in Netscape 4.04 and 4.05 [LaD98]. We eventually traced this bug down to the Netscape `SecurityManager` checking whether the fourth stack frame above it had a `ClassLoader`. The structure of the Netscape SecurityManager is such that the fourth stack frame above the check should belong to a constructor of a class that inherits from `java.lang.ClassLoader`. Although this check prohibits an applet from instantiating a subclass of `java.lang.ClassLoader`, `netscape.applet.AppletClassLoader`

[12]such as `readACL` and `writeACL`, see section 2.4.5.

is a public class with a public constructor that adds an additional stack frame, making the applet's constructor be the fifth stack frame, not the fourth, on the stack. If Netscape had used their stack inspection-based permission model for this security check, the problem would not have occurred. However, LaDue was unable to break the type system directly, because Netscape was identifying types correctly by (name, ClassLoader) pairs, not just names.

We generalized his result to instantiating any subclass of netscape.applet.AppletClassLoader. Unfortunately, AppletClassLoader implemented a final method for the upcall from the JVM to the ClassLoader, so we were unable to intercept name lookups. We overcame this difficulty by taking advantage of a bug in the AppletClassLoader implementation: it first called findLoadedClass, looking in the ClassLoader's internal hash table, before calling findSystemClass, to ask the NULL ClassLoader for a definition of a class. This meant that we could shadow the definition of any system class we chose, simply by defining it in our ClassLoader. The magic class to shadow was java.lang.Throwable, the base class of exceptions and errors.

The attack proceeds as follows: we took the Throwable class from JDK 1.1, and added an int and an Object field. We created a ClassLoader, and used it to load our version of Throwable. We used the same ClassLoader to load another class, Hack, of our creation, which had a static field of type RuntimeException[13]. From the main applet, loaded by an ordinary AppletClassLoader, we instantiate a subclass of RuntimeException, Kaboom, that we created with additional Object and int fields. Note that these fields are declared in the reverse order than the fields we added to our version of Throwable; the JVM's memory layout of objects is such

[13]We chose RuntimeException rather than Exception to avoid the need for a throws clause to make the Java compiler happy during the early stages of development. Because the verifier currently ignores the exception-throwing information in the bytecode, it is irrelevant.

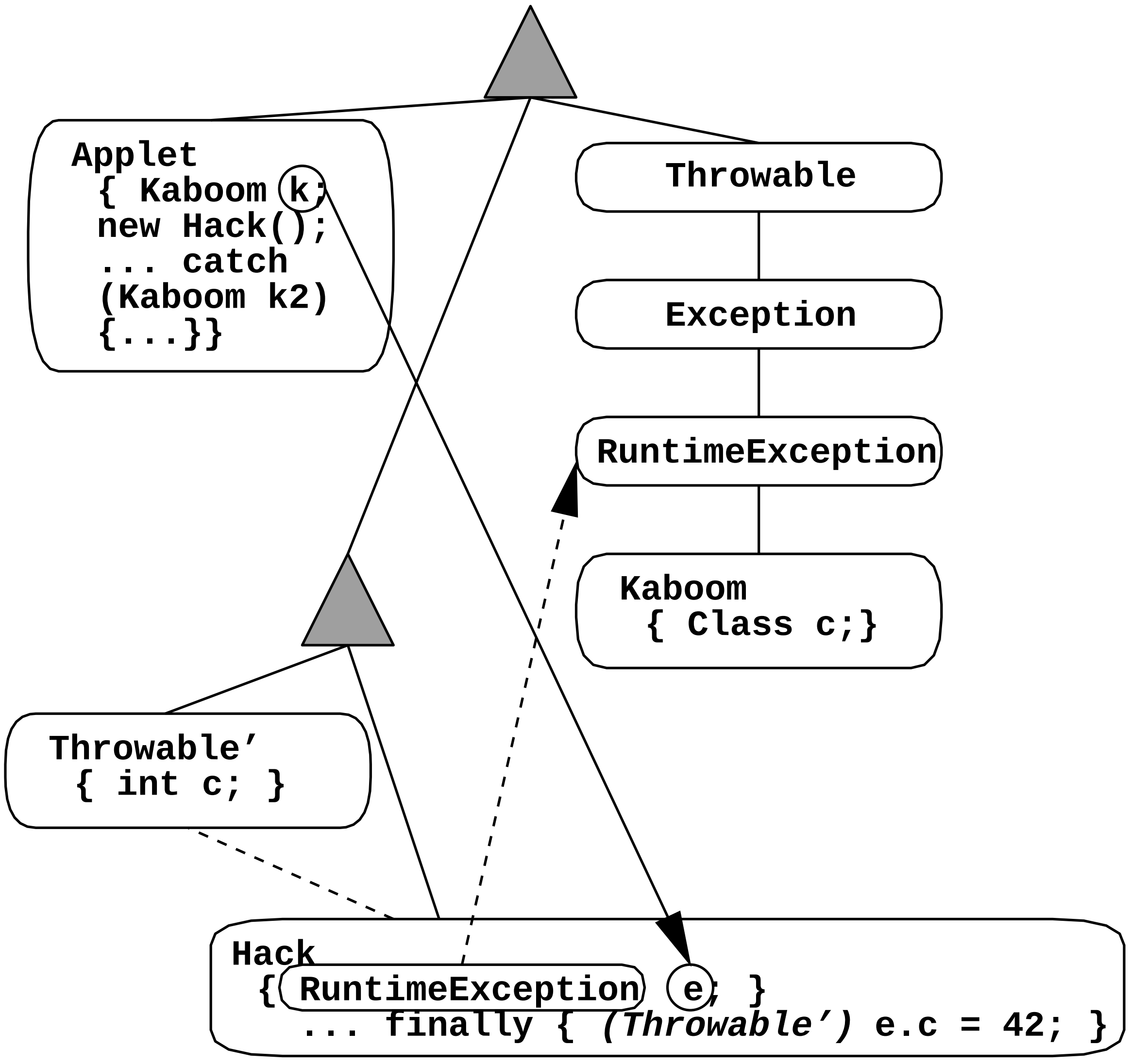

Figure 2.7: Type soundness failure of catch-any exception block. The shaded triangles represent `ClassLoaders`.

that the same memory location will have different types in different contexts. We then use the Java reflection API to store an instance of `Kaboom` into the static field of `Hack`. This completes the setup of the attack. See Figure 2.7.

The actual type soundness bug that we exploit cannot be expressed in Java, but is expressible in the JVM bytecode language. In order to compile Java's `try-finally` feature, there is a special wild card pattern that catches *any* exception, without requiring that it be a subclass of any nameable class. In this exception handler block, the verifier treats the exception value on the top of the stack as an instance of `Throwable` *as defined in the current* `ClassLoader`. The thought was that every thrown value is a subclass of `Throwable` (the verifier checks this at the throw site), but nothing more is known. Unfortunately, we've loaded two different definitions of `Throwable`, and the value we threw was an instance of a subclass of `Throwable` in the `NULL ClassLoader`, not an instance of `Throwable` in the current `ClassLoader`. Thus, we have a type soundness failure: the value of a variable is not a member of its type. Because the verifier lets us operate on this object according to the definition of `Throwable` in the current `ClassLoader`, we can store an integer into the `int` field that our version of `Throwable` declares. We then rethrow the object as normal, and catch it in our main applet class as its type, `Kaboom`. Its `Object` field has acquired the integer we stored into the "instance" of `Throwable`, completing the cast.

2.5 Java Language and Bytecode Weaknesses

The official Java and bytecode language *definitions* are weaker than they should be from a security viewpoint. The language has neither a formal semantics nor a formal description of its type system.[14] The module system is limited, the scoping rules are too liberal, and methods may be called on partially initialized objects [Has96]. The bytecode language is relatively difficult to verify, has no formal semantics[15], has unnaturally typed constructors, and does not enforce the `private` modifier on code loaded from the local file system. The separation of object creation and initialization poses problems. We explore these issues in more depth.

2.5.1 Language Weaknesses

The Java language definition [GJS96] has neither a formal semantics nor a formal description of its type system. We cannot formally reason about Java and the security properties of the Java libraries written in Java. Java lacks a formal description of its type system, yet the security of Java fundamentally relies on the soundness of its type system. Java's package system provides only basic modules, and these modules cannot be nested, although the name space superficially appears to be hierarchical. With properly nested modules, a programmer could limit the visibility of security-critical components. In the present Java system, only access to variables is controlled, not their visibility. Java also allows methods to be called from constructors: these methods may see a partially initialized object instance [Has96].

One nice feature of Java is that an object reference is roughly equivalent to a traditional capability [Lev84]. Because pointers cannot be forged, the possession of an object instance (such as an open file) represents the capability to use that file. However, the Java runtime libraries are not generally structured around using

[14]Drossopoulou and Eisenbach [DE97a], Syme [Sym97], and others are developing formal semantics for Java.

[15]A formal definition of the bytecode language is under development by Computational Logic, Inc. Stata and Abadi [SA98] have produced a type system for bytecode subroutines.

objects as capabilities. Used as capabilities, Java objects would have all the traditional problems of capability systems, e.g., difficulty tracking and controlling who has access to various system resources.

The Java language definition could be altered to reduce accidental leaks of information from public variables, and encourage better program structure with a richer module system than Java's `package` construct. Public variables in public classes are dangerous; it is hard to think of any safe application for them in their present form. Although Java's packages define multiple, non-interfering naming environments, richer interfaces and parameterized modules would be useful additions to the language. By having multiple interfaces to a module, a module could declare a richer interface for trusted clients, and a more restrictive interface for untrusted clients. The introduction of parameterized modules, like Standard ML's functors [MTH90], should also be investigated. Parameterized modules are a solution to the program structuring problem that opened up our man-in-the-middle attack (see section 2.4.3).

2.5.2 Bytecode Weaknesses

The Java bytecode language is where the security properties must ultimately be verified, as this is what gets sent to users to run. Unfortunately, it is rather difficult to verify bytecode. Bytecode is in a linear form and local variables can hold values of different types at different times, so type checking it requires global data flow analysis similar to the back end of an optimizing compiler [Yel95]; this analysis is complicated further by the existence of exceptions and exception handlers. Type checking normally occurs in the front end of a compiler, where it is a traversal of the abstract syntax tree [Pey87]. (The Juice system [FK97] works in the same way.) In the traditional case, type checking is compositional: the type correctness of a construct depends upon the current typing context, the type correctness of its subexpressions, and whether the current construct is typable by one of a finite set of rules. In the Java bytecode language, the verifier must show that all possible execution paths lead to the same virtual machine configuration — a much more complicated problem, and thus more prone to error. Recent work [SA98] on formalizing the semantics of the bytecode language is encouraging, but this should have been done in the design stage, not as an interesting research problem after the fact.

Object Initialization Creating and initializing a new object occurs in an interesting way: the object is created as an uninitialized instance of its class, duplicated on the stack, then its constructor is called. The constructor's type signature is *uninitialized instance of class* $\rightarrow$ *void*; it mutates the current typing context for the appropriate stack locations to initialized instances of their class. It is unusual for a *dynamic* function call to mutate the *static* typing context — in most statically typed languages, values have the same type throughout their life.

The initialization of Java objects seems unnecessarily baroque. The first time a class is used, its static constructors are executed. Then, for each instance of the class, a newly allocated object sets all of its instance variables to either null, zero, or false, as appropriate for the type of the variable. Then the appropriate constructor is called. Each constructor executes in three steps: First, it calls another constructor of its own class, or a constructor of its superclass. Next, any explicit initializers for instance variables (e.g. `int x = 6;`) written by the programmer are executed. Finally, the body of the constructor is executed. During the execution of a constructor body, the object is only partially initialized, yet arbitrary methods of the object may be invoked, including methods that have been overridden by subclasses, even if the subclasses' constructors have not yet run. Because Java's security partly relies on some classes throwing exceptions during initialization (to prevent untrusted code from creating an instance of a dangerous class), it seems unwise to have the system's security depend on programmers' understanding of such a complex feature.

Information Hiding We also note that the bytecode verifier does not enforce the semantics of the `private` modifier for bytecode loaded from the local file system. Two classes loaded from the local file system in the same package have access to all of each other's variables, whether or not they are declared `private`. In particular, *any* code in the `java.lang` package can set the system's security manager, although the definition of `System.security` and `System.setSecurityManager()` would seem to prevent this. The Java runtime allows the compiler to inline calls to `System.getSecurityManager()`, which may provide a small performance improvement, but with a security penalty for not enforcing the programmer's declared abstractions.

Extra Expressive Power The key distinction here is that the JVM is strictly richer than Java, that is, there are valid JVM programs (*i.e.*, bytecode that successfully verifies) that do not correspond to valid Java programs. Some examples:

1. The bytecode language has an intra-method GOTO statement, so arbitrary control-flow graphs may be formed. The Java language can produce only reducible flow graphs.

2. The bytecode language allows access to the exception value in an exception handler that catches *any* exception, whereas Java's `finally` block does not.

3. The bytecode language can catch an exception thrown by a super class constructor, which is impossible in Java. (The first statement in a constructor must be to call a constructor of its super class, or another constructor of the same class.) In early JVM releases, this could be used to (eventually) break the type system [DFW96].

The issue is how programmers reason about their code. They want to reason at the source code level about the possible behavior of hostile code.[16] But since the object code can contain contexts that cannot occur in the source code, *a priori*, there is no reason to believe that arbitrary semantic properties of Java will hold of the JVM bytecode language. The relationship between the JVM bytecode language and Java, and deriving sufficient, effectively checkable conditions on the bytecode to ensure that Java semantics are always upheld, is a major open problem in Java security today. Of course, complete formal specifications of both the JVM and Java are needed first.

2.6 Analysis

We found a number of interesting problems in an alpha version of HotJava, and various commercial versions of Netscape Navigator and Microsoft Internet Explorer. More instructive than the particular bugs we and others have found is an analysis of their possible causes. Policy enforcement failures, coupled with the lack of a formal security policy, make interesting information available to applets, and also provide channels to transmit it to an arbitrary third party. The integrity of the runtime system can also be compromised by applets. To compound these

[16]Noted by Andrew Appel in March 1996.

problems, no audit trail exists to reconstruct an attack afterward. In short, the Java runtime system is not a high assurance system.

2.6.1 Policy

The present documents on Netscape Navigator [Ros96], Microsoft Internet Explorer, and HotJava do not formally define a security policy. This contradicts the first of the Orange Book's Fundamental Computer Security Requirements, namely that "There must be an explicit and well-defined security policy enforced by the system." [Nat85] Without such a policy, it is unclear how a secure implementation is supposed to behave [Lan81]. In fact, Java has two entirely different uses: as a general purpose programming language, like C++, and as a system for developing untrusted applets on the Web. These roles require vastly different security policies for Java. The first role does not demand any extra security, as we expect the operating system to treat applications written in Java just like any other application, and we trust that the operating system's security policy will be enforced. Web applets, however, cannot be trusted with the full authority granted to a given user, and so require that Java define and implement a protected subsystem with an appropriate security policy.

2.6.2 Enforcement

The Java `SecurityManager` is intended to be a reference monitor [Lam71]. A reference monitor has three important properties:

1. It is always invoked.

2. It is tamperproof.

3. It is small enough to be verifiable.

Unfortunately, the Java `SecurityManager` design has weaknesses in all three areas.

- It is not always invoked: programmers writing the security-relevant portions of the Java runtime system must remember to explicitly call the `SecurityManager`. A failure to call the `SecurityManager` will result in access being granted, contrary to the security engineering principle that dangerous operations should fail unless permission is explicitly granted.
- It is not tamperproof: attacks that compromise the type system can alter information that the `SecurityManager` depends on.
- The `SecurityManager` code is the only formal specification of policies. Without a higher-level formal specification, informal policies may have incorrect implementations that go unnoticed. For example, the informal policies about network access were incorrectly coded in JDK 1.0 and Netscape Navigator 2.0's `SecurityManager` (see Section 2.4.3).

2.6.3 Integrity

The architecture of HotJava is inherently more prone than that of Netscape Navigator or Microsoft Internet Explorer to accidentally reveal internal state to an applet because the HotJava browser's state is kept in Java variables and classes. Variables and methods that are `public` are potentially very dangerous: they give the attacker a toe-hold into HotJava's internal state. Static synchronized methods and public instances of objects with synchronized methods lead to easy denial of service attacks, because any applet can acquire these locks and never release them. These are all issues that can be addressed with good design practices, coding standards, and code reviews.

Java's architecture does not include an identified trusted computing base (TCB) [Nat85]. Substantial and dispersed parts of the system must cooperate to maintain security. The bytecode verifier, and interpreter or native code generator must properly implement all the checks that are documented. The HotJava browser (a substantial program) must not export any security-critical, unchecked public interfaces. This does not approach the goal of a small, well defined, verifiable TCB. An analysis of which components require trust would have found the problems we have exploited, and perhaps solved some of them.

2.6.4 Accountability

The fourth fundamental requirement in the Orange Book is accountability: "Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party." [Nat85] The Java system does not define any auditing capability. If we wish to trust a Java implementation that runs bytecode downloaded across a network, a reliable audit trail is a necessity. The level of auditing should be selectable by the user or system administrator. As a minimum, files read and written from the local file system should be logged, along with network usage. Some users may wish to log the bytecode of all the programs they download. This requirement exists because the user cannot count on the attacker's Web site to remain unaltered after a successful attack. The Java runtime system should provide a configurable audit system.

2.7 Conclusion

We have seen many security failures in the Java implementations produced by Sun, Netscape, and Microsoft. Although the basic bugs are getting fixed over time, we are seeing the classic problem with the "penetrate and patch" methodology: new bugs get introduced with each new version of the software, and the software never converges to a bug-free state.

It is interesting to look back at this chapter in hindsight. Our complaints about the lack of any auditing still have not been addressed. Much work has been done on digital signatures, but support for managing signed code relies on the type-safety of the Java implementation, which we have been able to subvert in many implementations, including Netscape 4.05. HotJava source code is no longer generally available, so we have not examined HotJava 1.0 or 1.1 for vulnerabilities.

Chapter 3

Safe Dynamic Linking[1]

3.1 Introduction

We have seen that the design of dynamic linking is critical to the security of Java. Since Java is a (mostly) statically typed language [GJS96], if an applet can run in a different environment than the one in which it successfully passed the bytecode verifier, there is a potential security problem. We have 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 [DFWB97]. 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 chapter addresses the design of a type-safe dynamic linking system. Safe dynamic linking is not sufficient for building a secure system using language-based protection. However, linking should avoid breaking any language properties. The rest of the chapter is structured as follows. Section 3.2 discusses related

[1]An earlier version of this chapter appeared in the *Fourth ACM Conference on Computer and Communication Security* [Dea97].

work, Section 3.3 gives an informal statement of the problem, Section 3.4 informally discusses the problem, its ramifications, and solution, Section 3.5 discusses the formal treatment of the problem in PVS [OSR93], Section 3.6 briefly discusses implementation and assurance issues, and Section 3.7 concludes. The PVS specification is provided in an appendix.

3.2 Related Work

There has been 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 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 machine addresses. Static linking copies code (*e.g.*, the 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 [Org72], 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 [GLDW87] 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 it eliminates the storage

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

of multiple copies of the same library linked into programs on disk, and multiple processes can share the code (assuming it is not self-modifying), but not data areas, in memory. Each process gets its own data area, so the code does not have to be reentrant. 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 simply require the installation of the new libraries, and all dynamically linked programs on the system acquire the fix. Two different versions of a library that implement the same interface, but with 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. Although 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 an unsafe program: trying to interpret an integer as a pointer usually leads to a core dump. Because protection in Java depends on preventing users from forging object references, such a type mismatch would completely undermine the system.

[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.

Well-designed languages have module systems that provide support for separate compilation without these problems [Wir83, MTHM97]. C++ introduced *name mangling* as a way to encode type information into linking, to prevent inter-module type errors while still using standard linkers [Str94]. [4] Name mangling only prevents accidents; it does not prevent a malicious programmer from creating a type error.

The SPIN project at the University of Washington [BSP+95] and the Flux Project at the University of Utah [FBB+97] address dynamic linking for an operating systems viewpoint [SFPB96, OBLM93]. The SPIN work does not describe the mechanics of linking in detail; rather it focuses on access control via linking. Our work [WBDF97] with name space management is similar. The Flux work focuses on the implementation of a flexible and efficient dynamic linking mechanism; it does not discuss type safety. Other work at Utah [BLO94] has examined type safe linking for C, but does not address the problem faced in Java.

Appel and MacQueen [AM94] provide type-safe linking for Standard ML, and avoid the problems discussed here for two reasons. Standard ML is strictly statically scoped, and SML `structures` do not allow for recursion. Thus, loaded definitions can refer only to already existing definitions, and they never capture definitions loaded in the future.

Cardelli's recent work [Car97] 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

[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.

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 [Jan74] 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 [GLDW87] is very similar.) However, dynamic linking in Java *is* security-relevant, unlike Multics, where hardware-based rings were used for protection.

Drossopoulou and Eisenbach's recent work [DE97b] considers the type safety of a subset of Java. Although 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.

The situation in Java is different from the above situations. Java does not have type information available at link time; type checking (that is, bytecode verification) is interleaved with dynamic linking. Because 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 chapter considers the security-critical interaction of linking and type checking.

3.3 Informal Problem Statement

The Java runtime system may interleave type checking, linking, and program execution [GJS96]. The implementation from JavaSoft (and used by Netscape in their

Web browser) takes advantage of this freedom. Because 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 [Neu95].

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 [FM96], 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 `interfaces`.) 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 if and only if they have the same name. (In Java, two classes are the same if and only if they have the same name and the same classloader [GJS96].) Two interface types are the same if they are structurally equivalent. Interface types fit nicely in the *objects as records* model [Car88b], so we can define structurally equivalent types as having the same fields, where corresponding fields have the same type. For an implementation type Impl, we write $\mathit{Inter}(\mathit{Impl})$ for the corresponding interface type. The interested reader is referred to Fisher's thesis [Fis96] 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 $\Gamma = \{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}$, 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. $\Gamma(x) = \sigma$ if and only if $x : \sigma \in \Gamma$. Define $x_i \sqsubseteq x_j$, pronounced, "x_i is an interface subtype[e of x_j," if and only if $\mathit{Inter}(\sigma_i) = \mathit{Inter}(\sigma_j)$.[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 3.1 *If* $\Gamma \vdash M : \tau$ *and* $\Gamma \preceq \Gamma'$, *then* $\Gamma' \vdash M : \tau$.

The justification for this proposition can be found in [Mit90]; 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, *e.g.,* by adding new variables, classes, or methods to an existing program, without changing existing typing judgments in a type system that satisfies the proposition. A rigorous proof of this would require a formalization of the complete Java type system (see [DE97b] for work in this direction), and is beyond the scope of this thesis.

[5]The reader familiar with object-oriented type theory might expect the definition of $\sqsubseteq$ to be $\mathit{Inter}(\sigma_i) \leq \mathit{Inter}(\sigma_j)$. However, since Java variables declared as classes 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. In general, this relation will not be symmetric.

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.

In fact, we would ideally desire a stronger property: the author would like to know that the classes with the same name and interface (two properties that have been mechanically checked) behave the same, up to some isomorphism. Unfortunately, Java's type system is not sufficiently expressive to encode behavioral specifications, and we rely upon convention. While these conventions affect the behavior of programs (*e.g.,* consider swapping the create file and delete file operations), they do not affect the type safety of the system.

We introduce the following necessary restriction:

Restriction 3.2 (Linking) *A program can change its type context,* Γ*, to a new type context,* Γ'*, only 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 chapter considers the formalization and proof of this statement, along with the consequences of ignoring this limitation.

3.4 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 knows how to load and link code only 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 bytecode. (The bytecode language 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 model a specific implementation of `ClassLoader`. Although we do not care where the bytecode came from, we need the property that a `ClassLoader` returns only classes loaded by itself or the system `ClassLoader`, which we call the *primordial* `ClassLoader`. This is a provable theorem in our model, but we could easily insert a dynamic check into the trusted portion of the `ClassLoader` that would make this true for any `ClassLoader`. We assume that the `ClassLoader` hierarchy is a tree of height one (see Figure 3.1). JDK 1.0.2 and 1.1 meet these restrictions when running untrusted applets that cannot create their own `ClassLoaders`. Later, we shall see how to remove these limitations in a more general

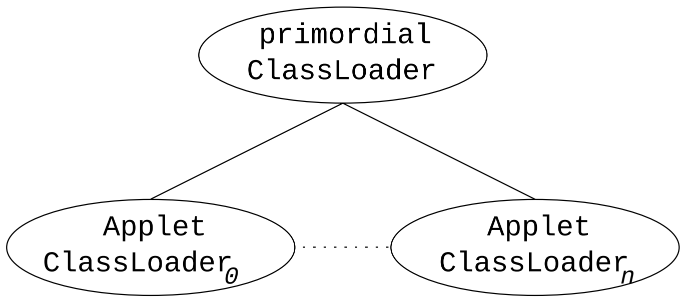

Figure 3.1: A diagram showing the `ClassLoader` hierarchy in a JVM.

model.

The original Java Development Kit (JDK) implementation (JDK 1.0.2) did not place *any* restrictions on the behavior of `ClassLoaders`. This led to the breakage of type safety, where integers could be used as object references, and vice versa [DFW96]. The type safety failure led to an untrusted applet being able to run arbitrary machine code, thus completely compromising the security of Java applets [DFWB97]. We discussed this issue with Sun, and the following language was added to the definition of Java [GJS96, Section 12.2, page 218]:

> A Java Virtual Machine system should maintain an internal table of classes and interfaces that have been loaded for the sake of resolving symbolic references. Each entry in the table consists of a fully qualified class name (as a string), a class loader, and Class object. Whenever a symbolic references to a class or interface is to be resolved, a class loader is identified that is responsible for loading the class or interface, if necessary. The table is consulted first, however; if it already contains an entry for that class name and class loader, then the class object in that entry is used and no method of the class loader is invoked. If the table contains no such entry, then the method loadClass of the class loader is invoked, giving it the name of the class or interface. If and when it returns, the class object that it returns is used to make a new entry in the table for that class name and class loader.
>
> The purpose of this internal table is to allow the verification process to assume, for its purposes, that two classes or interfaces are the same if they have the same name and the same class loader. This property allow a class to be verified without loading all the classes and interfaces

> that it uses, whether actively or passively. Well-behaved class loaders do maintain this property: given the same name twice, a good class loader should return the same class object each time. But without the internal table, an malicious class loader could violate this property and undermine the security of the Java type system. A basic principle of the Java language is that the type system cannot be subverted by code written in Java, not even by implementations of such otherwise sensitive system classes as ClassLoader and SecurityManager.

Code to implement this restriction (essentially the same as Restriction 3.2) is part of JDK 1.1.

However, the specified restriction applies only to a single `ClassLoader`; multiple `ClassLoaders` can collude to break the type system in the absence of the limitation that a `ClassLoader` can return only a class defined by itself or the primordial `ClassLoader`, or if the `ClassLoader` hierarchy has a height greater than one. See Chapter 4 for a discussion of the underlying problem and its solution.

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. For example, the first time a `ClassLoader` is asked to resolve the name `Alpha`, it could return a pointer to the class:

   ```
   class Alpha { public Object x; }
   ```

   The second time it is asked to resolve `Alpha`, it could return a pointer to the class:

   ```
   class Alpha { public int x; }
   ```

This confuses the type system so that at runtime an integer is assigned to an object reference, thereby breaking the system. In Java runtimes derived from JDK 1.0.x releases, including 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 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, and requires that the type context be a consistent extension of the type context defined by the primordial `ClassLoader`. It shows that all relevant operations 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.

3.5 Formal Treatment in PVS

PVS [OSR93][6] is the PROTOTYPE VERIFICATION SYSTEM, a current SRI research development in formal methods and theorem proving. PVS has been used in many different verification efforts, including a microprocessor [SM95], floating point division [RSS96], fault-tolerant algorithms [LR93], and multimedia frameworks [RRV95], by users at SRI and other sites. PVS combines a specification language with a variety of theorem proving tools.

Proposition 3.1 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 Restriction 1. This chapter 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 specification that looks very much like a functional program, along with a correctness proof. Whereas 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.

3.5.1 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

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

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. This correctness proof does not imply the correctness of any implementation of Java. This model merely shows that dynamic linking *can* peacefully co-exist with static typing. The remainder of this section is meant to provide commentary for the PVS specification in Appendix A.

PVS Types

The core structure in the model is the `ClassTable`, which contains two mappings: the first, an *environment* mapping `(Name, ClassLoader)` pairs to `ClassIDs`, and the second, a *store* mapping `ClassIDs` to `Class` objects.

$$\boxed{\begin{aligned}
&\text{ClassID} : \text{TYPE} = \text{Ident}\\
&\text{ClassList} : \text{TYPE} = \text{list[Class]}\\
&\text{ClassIDMap} : \text{TYPE} = \text{FUNCTION}[\text{ClassID} \to \text{Class}]\\
&\text{ClassDB} : \text{TYPE} = [\text{ClassID}, \text{ClassIDMap}]\\
&\text{EnvEntry} : \text{TYPE} = [\text{string}, \text{ClassLoader}, \text{list[ClassID]}]
\end{aligned}}$$

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 `Classes` exists so that linking does not have to change the environment; it just 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. We keep a mapping from a `(Name,ClassLoader)` pair to a list of `ClassIDs`; 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 ClassIDs instead of a set, so we can tell what order things happened in if anything should ever break. We define a state as safe if and only if each (Name, ClassLoader) pair maps to at most one ClassID. We also prove that we never delete a mapping; otherwise our partial function could vary over time, which is exactly the behavior we seek to prohibit.

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


ValidClass((c : ClassBase)) : bool =
 CASES c OF
  unresolved(n, r, l) : TRUE,
  resolved(n, r, loader, links) :
    (∀ (cl : ClassBase) :
      (cl ∈ links) ⊃
        loader(cl) = loader(c) ∨ loader(cl) = primordialClassLoader)
  ENDCASES
```

The definition of Class and ClassTable use a powerful feature of PVS, predicate subtypes. ClassBase is a datatype that represents a Java class in our model. However, to be a valid Java class, it must also satisfy the ValidClass predicate. While PVS can decide whether a variable is of type ClassBase, it cannot, in general, decide whether a variable is of type Class. We must prove the type correctness condition (TCC), *i.e.*, that ValidClass is satisfied, for any value we bind to a variable of type Class. We must also prove that the return value of any function declared to return a Class satisfies the ValidClass predicate. Fortunately, we have the full power of PVS available for these proofs.

```
ClassTableBase : TYPE  = [# env : list[EnvEntry], store : ClassDB#]

ValidCT((ctb : ClassTableBase)) : bool =
  (∀ (e : EnvEntry) :
    (e ∈ env(ctb)) ⊃
      LET y = PROJ_3(e)
        IN
          every(λ (x : ClassID) :
              PROJ_2(e) = loader(PROJ_2(store(ctb))(x))∧
                x ≤ PROJ_1(store(ctb)),
            y))

ClassTable : TYPE  = {ctb : ClassTableBase | ValidCT(ctb)}
```

ClassTableBase is a record type with two fields, env and store. A value is a ClassTable if and only if it is a ClassTableBase record, and it satisfies the ValidCT predicate. As above, we must prove TCCs for any values we declare to be ClassTables. The ValidCT predicate has two conjuncts: the first conjunct says that all (Name, ClassLoader) pairs map to classes that have the same ClassLoader, and the second conjunct says that the ClassIDs that the (Name, ClassLoader) pairs are mapped by the environment to ClassIDs that are less than or equal to the next ClassID to be allocated.[7] The need for the second conjunct is not intuitively obvious; it is an invariant strengthening that is needed to make the induction go through.

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 conveniently handle the mutual recursion found in the Java implementation. Since our model uses the ClassLoader only 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 unresolved symbols,

[7]We are taking advantage of the concrete representation of ClassIDs as integers.

but no pointers to other classes. The unresolved constructor is required because 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 would 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 the class this object is an instance of. Although this representation is fairly abstract, it suffices for our proofs.

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 `FindClassIDswCL` function takes a `ClassTable`, the name of a class, and the requested `ClassLoader`, and returns a list of `ClassID`s and a `ClassLoader`. The predicate subtype on `FindClassIDswCL` ensures that the returned `ClassID`s map to classes (in the store) that have the requested `ClassLoader`. `FindClass` applies the current store, mapping `ClassID`s to `Class`es, to the result of `FindClassIDs`, which is simply a projection from `FindClassIDswCL`.

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 called only after the appropriate safety checks have been made. We implement `define` in terms of a helper function, `InsertClass`. 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.

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`. This ensures that any `ClassLoader` provides a consistent extension of the runtime system, as defined by the primordial `ClassLoader`. 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, we tell PVS that some external references exist by using 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 returns only “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`. We implement `resolve` in terms of a helper function, `ReplaceClass`. The `ReplaceClass` function takes a `ClassTable`, the old and new classes, and the appropriate `ClassLoader`, and updates the store if and only if the appropriate class is found. It then returns the new `ClassTable`. If no appropriate class is found, it returns the unchanged `ClassTable`. Note that `resolve` does not change the `ClassLoader` of the resolved class.

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.

3.5.2 The Proofs

This chapter offers two contributions. Although Restriction 3.2 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 general: any language whose type system satisfies Proposition 1 (and most do) can use the results of this chapter. Given an operational semantics for the language under inspection, a completely formal safety proof can be constructed. Drossopoulou and Eisenbach's work [DE97b] is a good beginning, but was not available when this work began. The second contribution is a proof that the requirements of Restriction 1 are satisfied by our model. Here the proofs are discussed at a high level; PVS takes care of the details.

There are nine lemmas, thirty-four type correctness conditions, and eleven theorems which establish the result. Formal proof of these theorems increases our confidence in the correctness of the specification. Twenty-six of the TCCs are automatically handled by PVS, and three more are trivial. The eleven 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. Because 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. The proofs are long and not very enlightening, so here we present only brief outlines of the proofs, with the mechanized proof of `loadClass_inv` in Appendix B. The details are all routine, and handled by PVS.

Lemmas

We define nine lemmas which encapsulate useful facts about our model. PVS does not distinguish between lemma and theorem, so the distinction is arbitrary. We call the main results theorems, and the rest lemmas, to help guide the reader.

every_monotone This is the simple fact that if $\forall x.P(x) \rightarrow Q(x)$, then if we have a list, all of whose elements satisfy $P(x)$, then all the elements of the list also satisfy $Q(x)$. The proof is by induction on the length of the list.

```
every_monotone : LEMMA
  (∀ (p, q : PRED[ClassID]), (y : list[ClassID]) :
    (∀ (x : ClassID) : p(x) ⊃ q(x)) ∧ every(p, y) ⊃ every(q, y))
```

every_FindClassIDswCL This lemma states that all of the classes returned by `FindClassIDswCL` have the same `ClassLoader`, and that the `ClassLoader` is the one passed as an argument to `FindClassIDswCL`. Furthermore, this is also true in a new store, where a new `ClassID` is bound to a new class. Here we see the second conjunct of the predicate `ValidCT` coming into play.

```
every_FindClassIDswCL : LEMMA
  (∀ (cldr : ClassLoader, ct : ClassTable, nm : string, refs : list[string]) :
    every(λ (x : ClassID) :
              cldr =
                loader(PROJ_2(store(ct))
                   WITH [(1 + PROJ_1(store(ct))) :=
                     unresolved(nm,
                       refs, cldr)](x))∧
               x ≤ 1 + PROJ_1(store(ct)),
             PROJ_1(FindClassIDswCL(ct, nm, cldr))))
```

linkClass_loader_inv This lemma states that the `loader` field of a class stays the same after the class is linked. The proof proceeds by induction on the number of unresolved references to be linked.

```
linkClass_loader_inv : LEMMA (∀ ct, cl : loader(cl) = loader(PROJ_1(linkClass(ct, cl))))
```

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`.

$$\boxed{\begin{array}{l}\text{MapPreservesLength : LEMMA}\\ \quad (\forall\,(f : \text{FUNCTION}[\text{ClassID} \to \text{Class}]), (l : \text{list}[\text{ClassID}]) : \text{length}(\text{map}(f, l)) = \text{length}(l))\end{array}}$$

proj1_FindClassIDswCL 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 in the environment.

$$\boxed{\begin{array}{l}\text{proj1\_FindClassIDswCL : LEMMA}\\ \quad (\forall\,(\text{ct} : \text{ClassTable}), (\text{nm} : \text{string}), (\text{cldr} : \text{ClassLoader}), (\text{classdb} : \text{ClassDB}) :\\ \quad\quad \text{ValidCT}((\#\text{env} := \text{env}(\text{ct}), \text{store} := \text{classdb}\#)) \supset\\ \quad\quad\quad \text{FindClassIDswCL}((\#\text{env} := \text{env}(\text{ct}), \text{store} := \text{classdb}\#), \text{nm}, \text{cldr}) =\\ \quad\quad\quad\quad \text{FindClassIDswCL}(\text{ct}, \text{nm}, \text{cldr}))\end{array}}$$

proj1_FindClassIDs This lemma is very similar to `proj1_FindClassIDswCL`, but it considers only the `ClassIDs`, not the `ClassLoaders`. The proof is immediate from `proj1_FindClassIDswCL`.

[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 : LEMMA
  (∀ (ct : ClassTable), (nm : string), (cldr : ClassLoader), (classdb : ClassDB) :
    ValidCT((#env := env(ct), store := classdb#)) ⊃
      FindClassIDs((#env := env(ct), store := classdb#), nm, cldr) =
        FindClassIDs(ct, nm, cldr))
```

define_mono This lemma states that defining a new class does not remove any class bindings from the environment. The proof follows from the definition of `define`, the `ClassTable` satisfying `ValidCT`, and the `every_monotone` lemma.

```
define_mono : LEMMA (∀ ct, nm, cldr : Monotonic(ct, PROJ_2(define(ct, nm, Input, cldr))))
```

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. Because safety is a function of the environment, not the store, this is intuitively obvious. The proof uses the `MapPreservesLength` lemma.

```
safe_proj : LEMMA
  (∀ ct, (mapping : ClassIDMap) :
    Safe(ct) ∧ ValidCT((#env := env(ct), store := (PROJ_1(store(ct)), mapping)#)) ⊃
      Safe((#env := env(ct), store := (PROJ_1(store(ct)), mapping)#)))
```

Resolve This lemma 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.

```
Resolve : LEMMA (∀ (cl : Class), (ct : ClassTable) :
references(PROJ_1(linkClass(ct, cl))) = null)
```

Type Correctness Conditions

We shall discuss only the five non-trivial TCCs here. Twenty-six TCCs are automatically proven by PVS without manual assistance. Two TCCs, which show the existence of members of a type (to satisfy the declaration of a non-empty type), merely require the user to instantiate a variable with a suitable term. The third trivial TCC is that an empty `ClassTable` satisfies the `ValidCT` predicate; this is vacuously true. The proofs of the five remaining TCCs required significant manual intervention.

FindClassIDswCL_TCC4 This TCC requires us to show that the class we return has the specified `ClassLoader`. The result follows from the `every_monotone` lemma, coupled with the definition of `ValidCT`.

```
 FindClassIDswCL_TCC4: OBLIGATION
(FORALL (hd: EnvEntry, tab, tl: list[EnvEntry],
         v: [d: [ClassTable, string, ClassLoader]
             -> ll_ldr: [list[ClassID], ClassLoader]
             | PROJ_2(ll_ldr) = PROJ_3(d)
             AND
             every(LAMBDA (x: ClassID):
                   loader(PROJ_2(store(PROJ_1(d)))(x)) = PROJ_3(d),
                   PROJ_1(ll_ldr))],
         cldr: ClassLoader, ct: ClassTable, nm: string):
 env(ct)
 =
 cons[[string, ClassLoader,
       x: list[number]
       |
       every(LAMBDA (x: number):
             real_pred(x)
             AND rational_pred(x)
             AND integer_pred(x) AND x >= 0)(x)]](hd,
                                                  tl)
 AND tab = env(ct)
 IMPLIES
 PROJ_2(IF PROJ_1(hd) = nm AND PROJ_2(hd) = cldr
        THEN (PROJ_3(hd), PROJ_2(hd))
        ELSE v((# env := tl, store := store(ct) #), nm, cldr)
        ENDIF)
 = cldr
 AND
 every[ClassID](LAMBDA (x: ClassID): loader(PROJ_2(store(ct))(x)) = cldr,
               PROJ_1(IF PROJ_1(hd) = nm AND PROJ_2(hd) = cldr
                      THEN (PROJ_3(hd), PROJ_2(hd))
                      ELSE v((# env := tl, store := store(ct) #),
                             nm, cldr)
                      ENDIF)));
```

define_TCC1 This TCC requires us to show that defining a new class results in a ClassTable that satisfies ValidCT. After simplification, there are two cases to consider: The first case concerns any previous (Name,ClassLoader) to Class mappings, with the same (Name,ClassLoader) pair as the definition we are considering. This follows from the every_FindClassIDswCL lemma. The second case concerns the new definition. The result follows from the definition of define and the every_monotone lemma.

```
  define_TCC1: OBLIGATION
(FORALL (cl, cldr: ClassLoader, ct: ClassTable,
        nm: string, refs: list[string]):
 cl = mkClass(nm, refs, cldr)
 IMPLIES
 ValidCT((# env :=
            cons[[string, ClassLoader,
                  list[ClassID]]]((nm,
                                   cldr,
                                   cons[Ident]
                                   (GetNextID
                                    (PROJ_1(store(ct))),
                                    FindClassIDs(ct,
                                                 nm, cldr))),
                                  env(ct)),
            store :=
            (GetNextID(PROJ_1(store(ct))),
             PROJ_2(store(ct))
             WITH [(GetNextID(PROJ_1(store(ct))))
                   := mkClass(nm, refs, cldr)])
            #)));
```

linkClass_TCC1 This TCC requires us to show that each step of linking a class preserves the `ValidClass` predicate. Recall that the `ValidClass` predicate requires all classes referenced by a given class have either the same `ClassLoader` as the given class, or were loaded by the `primordialClassLoader`. The key step in the proof is to use the type information about the return value of `FindClassIDswCL` – we know that it returns a class loaded by the given `ClassLoader`, or the `primordial-ClassLoader`. In conjunction with the definition of `loadClass`, the result follows after simplification.

```
  linkClass_TCC1: OBLIGATION
(FORALL (getClass, hd: string, newCl, newCt, res,
        tl: list[string], cl: Class, ct: ClassTable):
 getClass = (LAMBDA (n: string): loadClass(ct, n, loader(cl)))
 AND references(cl) = cons[string](hd, tl)
 AND res = PROJ_1(getClass(hd))
 AND newCt = PROJ_2(getClass(hd))
 AND newCl
 = CASES cl OF
 unresolved(name, references, loader):
 resolved(name, tl, loader, cons[Class](res, null[Class])),
 resolved(name, references, loader, linked):
 resolved(name, tl, loader, cons[ClassBase](res, linked))
 ENDCASES
 IMPLIES ValidClass(newCl));
```

linkClass_TCC3 This TCC, together with `linkClass_TCC1`, forms an inductive proof that `linkClass` preserves the validity of any class. If we have a class that has not yet been linked, but has no unresolved references, the result of `linkClass` vacuously satisfies the `ValidClass` predicate. If the class has already been linked, it is immediately returned. Because we were given a valid class by assumption, the result is also valid. Otherwise we must consider the recursive case. Because TCCs occur logically before the definition in PVS, the definition is not in scope. PVS models the recursive call by universally quantifying over a function with the same type as the recursive function. The proof proceeds by using the type of the universally quantified function. Some simplification and an appeal to the `closedWorld` lemma establish the result.

```
  linkClass_TCC3: OBLIGATION
(FORALL (getClass, v: [[ClassTable, Class] -> [Class, ClassTable]],
        cl: Class, ct: ClassTable):
 getClass = (LAMBDA (n: string): loadClass(ct, n, loader(cl)))
 IMPLIES
 ValidClass(PROJ_1(CASES references(cl) OF
                  null:
                  IF unresolved?(cl)
                  THEN
                  (resolved(name(cl),
                            null[string],
                            loader(cl),
                            null[ClassBase]),
                   ct)
                  ELSE (cl, ct)
                  ENDIF,
                  cons(hd, tl):
                  v
                  (PROJ_2
                   (loadClass(ct,
                              hd, loader(cl))),
                   CASES cl OF
                   unresolved(name,
                   references,
                   loader):
                   resolved(name,
                            tl, loader,
                            cons[Class]
                            (PROJ_1
                             (loadClass(ct,
                                        hd, loader(cl))),
                             null[Class])),
                   resolved(name,
                   references,
                   loader, linked):
                   resolved(name,
                            tl, loader,
                            cons[ClassBase]
                            (PROJ_1
                             (loadClass(ct,
                                        hd, loader(cl))),
                             linked))
                   ENDCASES)
                  ENDCASES)));
```

resolve_TCC1 This TCC requires us to show that linking preserves the validity of a `ClassTable`, namely, that replacing a class in the store with its resolved version results in a `ClassTable` that satisfies the `ValidCT` predicate. This is where the strengthening of the `ValidCT` predicate, requiring that every `ClassID` point to an already allocated element of the store, is used. The proof proceeds by a case split on the result of looking up an arbitrary class after linking it. If the result is the empty list, the TCC follows immediately from the type of linkClass, which has already been shown to return a ClassTable that satisfies ValidCT. Otherwise, we note that linkClass returns a valid ClassTable, and the linkClass_loader_inv lemma (linking does not change a class's ClassLoader). At this point, an appeal to the every_monotone lemma, combined with the type constraint on the return value of FindClassIDswCL establishes the result.

```
  resolve_TCC1: OBLIGATION
(FORALL (newCl, newCt, cl: Class, ct: ClassTable):
 newCl = PROJ_1(linkClass(ct, cl)) AND newCt = PROJ_2(linkClass(ct, cl))
 IMPLIES
 ValidCT(CASES FindClassIDs(PROJ_2(linkClass(ct, cl)),
                          name(cl), loader(cl)) OF
         cons(hd, tl):
         (# env :=
            env
            (PROJ_2
             (linkClass(ct,
                        cl))),
            store :=
            (PROJ_1
             (store
              (PROJ_2
               (linkClass(ct,
                          cl)))),
             PROJ_2
             (store
              (PROJ_2
               (linkClass(ct,
                          cl))))
             WITH [hd :=
                   PROJ_1
                   (linkClass(ct,
                              cl))])
            #),
         null:
         PROJ_2
         (linkClass(ct,
                    cl))
         ENDCASES));
```

Theorems

There are three classes of theorems: safety theorems, monotonicity theorems, and two miscellaneous theorems. The safety theorems assert that each ClassTable produced has at most one `ClassID` per `(Name, ClassLoader)`-pair. The monotonicity theorems assert that `(Name, ClassLoader)` to `ClassID` mappings are never deleted. The miscellaneous theorems are `closedWorld` and `consistExt`, which assert that a `ClassLoader` never returns a class defined by another `ClassLoader` (other than the primordial `ClassLoader`), and that every `ClassLoader` is a consistent extension of the primordial `ClassLoader`.

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.

$$\boxed{\text{Initial\_Safe} : \text{THEOREM Safe(sysClassTable)}}$$

loadClass_inv This is the first case to consider in proving the safety 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`.

$$\boxed{\begin{array}{l}\text{loadClass\_inv} : \text{THEOREM } (\forall\, \text{ct}, \text{nm}, \text{cldr} : \\ \qquad\qquad\qquad \text{Safe(ct)} \supset \text{Safe(PROJ\_2(loadClass(ct}, \text{nm}, \text{cldr))))}\end{array}}$$

loadClass_mono This monotonicity proof assets that `loadClass` does not delete any mappings from the environment. The proof proceeds by a case-split on whether the appropriate class is already loaded. If so, the final environment is the same as the starting environment. Otherwise, `define` is called, and `define_mono` asserts the desired result.

$$\boxed{\begin{array}{l}\text{loadClass\_mono} : \text{THEOREM } (\forall\, \text{ct}, \text{nm}, \text{cldr} : \\ \qquad\qquad\qquad \text{Monotonic(ct}, \text{PROJ\_2(loadClass(ct}, \text{nm}, \text{cldr))))}\end{array}}$$

linkClass_inv This case of the invariant states that `linkClass` preserves safety. The intuitive idea is that `linkClass` modifies only 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.

$$\boxed{\text{linkClass\_inv}: \text{THEOREM } (\forall\, ct, cl: \text{Safe}(ct) \supset \text{Safe}(\text{PROJ\_2}(\text{linkClass}(ct, cl))))}$$

linkClass_mono This monotonicity proof assets that `linkClass` does not delete any mappings from the environment. The proof proceeds by induction on the number of resolved references. There is a case-split to handle the two different cases of linking a class with no unresolved references (either because the class has already been linked, or the class is `Object`). The induction step follows from `loadClass_mono`.

$$\boxed{\text{linkClass\_mono}: \text{THEOREM } (\forall\, ct, cl: \text{Monotonic}(ct, \text{PROJ\_2}(\text{linkClass}(ct, cl))))}$$

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

$$\boxed{\text{forName\_inv}: \text{THEOREM } (\forall\, ct, nm, cldr: \text{Safe}(ct) \supset \text{Safe}(\text{PROJ\_2}(\text{forName}(ct, nm, cldr))))}$$

forName_mono This monotonicity proof asserts that the `forName` operation does not delete any mappings from the environment. Because `forName` either returns the given environment or calls `loadClass`, the result is immediate.

$$\boxed{\text{forName\_mono}: \text{THEOREM } (\forall\, ct, nm, cldr: \text{Monotonic}(ct, \text{PROJ\_2}(\text{forName}(ct, nm, cldr))))}$$

Resolve_inv This is the last case of the safety 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.

$$\boxed{\text{Resolve\_inv}: \text{THEOREM } (\forall\, ct, cl, cldr: \text{Safe}(ct) \supset \text{Safe}(\text{resolve}(ct, cl, cldr)))}$$

Resolve_mono This final monotonicity proof asserts that the `resolve` operation does not delete any mappings from the environment. The desired result follows from the monotonicity of `linkClass`, the type correctness of `resolve`, and `proj1_FindClassIDs`.

$$\boxed{\text{Resolve\_mono}: \text{THEOREM } (\forall\, ct, cl, cldr: \text{Monotonic}(ct, \text{resolve}(ct, cl, cldr)))}$$

closedWorld This theorem asserts that a `ClassLoader` will load only classes that were defined by itself or the primordial `ClassLoader`. Intuitively, this follows from the definition of `loadClass, findSysClass`, and `FindClass`. The formal proof requires the predicate subtype on `FindClassIDswCL`; with that, it is almost immediate.

$$\boxed{\begin{array}{l}\text{closedWorld}: \text{THEOREM}\\ \quad(\forall\, ct, nm, cldr:\\ \qquad \text{LET classloader} = \text{loader}(\text{PROJ\_1}(\text{loadClass}(ct, nm, cldr)))\\ \qquad\ \ \text{IN classloader} = \text{cldr} \lor \text{classloader} = \text{primordialClassLoader})\end{array}}$$

consistExt This theorem asserts that every `ClassLoader` is a consistent extension of the primordial `ClassLoader`. The proof follows immediately from definition of `loadClass`.

```
consistExt : THEOREM
  (∀ ct, nm, cl, cldr :
     cons?(findSysClass(ct, nm)) ⊃
       car(findSysClass(ct, nm)) = PROJ_1(loadClass(ct, nm, cldr)))
```

The last two proofs are important for the overall type safety of a JVM. These properties, with the additional assumptions that there are no other `ClassLoaders` in the system, and the runtime system libraries are closed (*i.e.,* do not mention any classes that they do not define), are sufficient to prevent a type safety breach across `ClassLoader` boundaries. If we wish to allow arbitrary, user-written `ClassLoaders`, then trusted code needs to make dynamic checks to ensure that the appropriate generalizations of `closedWorld` and `consistExt` hold. The problem that must be solved is passing an object across a `ClassLoader` boundary, where the object is treated as having a different type (because there are two classes with different definitions) in the type context defined by each `ClassLoader`.

3.6 Implementation and Assurance

This chapter presents a *model* of dynamic linking, and proves a safety property under one assumption. While this is a nice result, systems in the real world are 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. Because 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.

There are two conclusions for implementors: each class definition must be loaded *exactly* once for each classloader, and linking across `ClassLoader` boundaries must be carefully controlled. The simplest way to ensure that each class is defined at most once per `ClassLoader` is for the runtime system to track which classes have been loaded by which classloaders and ask a classloader to provide the definition of a class only once. We assume that a classloader will either provide a class or fail consistently. In this chapter, we do not allow untrusted `ClassLoaders`, so we can prove properties about the behavior of all the `ClassLoaders` in the system. Although it would be simple to extend this model to allow untrusted `ClassLoaders` by wrapping the untrusted code with dynamic checks of the properties we have statically proven about our `ClassLoader` model, in the next chapter we examine an alternative model instead.

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 five pages. Our proofs were performed with lists, because they admit simple inductive proofs. 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.

3.7 Conclusion

This chapter presents one of many models for dynamic linking. We present a formal proof to show that dynamic linking need not interfere with static type checking. Although 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.

Chapter 4

Linking and Scope

Much progress has been made in producing formal definitions of Java, *e.g.*, Drossopoulou and Eisenbach [DE97b] and Syme [Sym97]. These works, however, both assume that the environment is well formed, in particular, that each class is defined at most once. In the previous chapter, we showed how to ensure that condition, and strengthened it to preclude other unsound cases that can arise with multiple environments.

In this chapter, we discuss the root of the problem ignored by the present formal specifications of Java [Sym97, DE97b]: *Classes have dynamic scope in JVM*. Given the well-known failure of static typing with dynamic scope at the value level [Mit96, page 60], it should not be surprising that dynamic scoping at the type level induces a similar failure. The rest of this chapter discusses related work (Section 4.1), explains our choice of PCF (Section 4.2), models it in two PCF variants, PCF$_\tau$ (Section 4.3) and PCF$_{\tau,\rho}$ (Section 4.4), proposes a solution to the problem (Section 4.5), proves results for the PCF$_{\tau,\rho}$ language (Section 4.5.1), relates the PCF formulation of the problem back to Java (Section 4.6), and concludes (Section 4.7).

4.1 Related Work

We chose to build a model of JVM making extensive use of environments, rather than dynamic linking, as in Cardelli [Car97] or Chapter 3. We did this because the bindings must be immutable in JVM. In our earlier work, we implemented this immutability by introducing a store, with the environment mapping class names to store locations, and the store mapping locations to classes. Although this worked out technically, the proofs became much more difficult. Cardelli also did not consider recursive definitions. Although it would be easy to extend his work in that direction, the added necessity to do so further pushed us towards the environment model given here. Cardelli observed that environments are related to his linksets [Car97, Section 2.1].

Our presentation of type dynamic has followed Abadi, *et al.* [ACPP91], and Leroy and Mauny [LM93]. These were the most useful formulations for our needs. Henglein [Hen94] attempts to minimize the amount of tagging required; because Java requires type tags for other reasons, this is not an issue for us.

First-class environments have a long history in reflective languages [WF88]. Java's ClassLoaders can be considered a reflection mechanism; they control environment lookup in the language itself. The JDK 1.1 Reflection API adds more reflective capabilities to Java, but they are special methods that do not interact with the normal execution environment. Lee and Friedman's "Quasi-static scope" [LF93] is perhaps the closest system to ours, but its values require special accessors. Most work on first-class environments has concentrated on applications of the concept, whereas we are using the concept to model something that already exists in Java. To the best of my knowledge, all previous work on first-class environments has occurred in dynamically typed languages (usually a variant of Lisp).

The Pebble language [BL84] has a more sophisticated version of our `binding` types. In Pebble, bindings have types that depend upon their values, whereas $\text{PCF}_{\tau,\rho}$ bindings are either of type $\texttt{binding}_{\tau}$ or $\texttt{binding}_{\nu}$. Our type system is much simpler than Pebble's; we omit dependent types and $\mathsf{type} : \mathsf{type}$. In Pebble, Burstall and Lampson note the same phase distinction problem that we do.

Liang and Bracha's OOPSLA98 paper [LB98], describes Sun's implementation of safe class loading in Java 2.0 (formerly known as JDK 1.2). Their design was done after discussions about an earlier version of this chapter.

4.2 Modeling the Problem in PCF

Instead of explaining the details of dynamic linking in Java at this point, we present examples in variants of PCF [Sco93]. By defining little languages that focus on the areas we want to study, we dramatically simplify our proofs. We also are able to avoid a detailed discussion of `ClassLoaders` until the end of the chapter. The main results do not refer to Java.

The problem with dynamic scope, of course, is the classical problem of free variable capture. The first example shows the capture of a redefinition of a variable, using a simple extension of PCF. The second example considers PCF extended with multiple environments, and shows another aspect of free variable capture, which also leads to unsoundness. The first example is not a language proposal; it merely illustrates the problem in a simple, familiar setting. In the second example, we provide sufficient restrictions to prove soundness.

$$\begin{array}{rcl} E & ::= & c \mid x \mid \lambda x{:}\tau.E \mid (E_1\ E_2) \mid \texttt{let val}\ x{:}\tau = E_1 \texttt{in}\ E_2 \mid \texttt{let type}\ t\ = \tau\ \texttt{in}\ E \\ \tau, \sigma & ::= & \texttt{int} \mid \texttt{bool} \mid \tau\ \rightarrow\ \sigma \mid t \end{array}$$

Figure 4.1: PCF$_\tau$ definition

4.3 PCF$_\tau$

We begin by defining a variant of simply typed, call-by-value PCF that we call PCF$_\tau$: The one unusual aspect of PCF$_\tau$ is the `let type` construction. We call the bound variable in a `let type` a *type identifier*, to distinguish it from a polymorphic type variable. Type identifiers have dynamic scope; variables have lexical scope. We will define this language by the translation $\mathcal{E}[\![]\!]$ to a variant of the second-order λ-calculus [Rey74], where type variables have dynamic scope, and with the addition of `int` and `bool` types. Because type identifiers are dynamically scoped, we do *not* use capture-avoiding substitution for them. Regular variables do use capture-avoiding substitution. As usual, we denote value application with (), and type application with []. We consider α-equivalent terms congruent.

The typing rules are found in Figure 4.3. Type equality is purely syntactic: two type identifiers are equal if and only if they have the same name, and function types are equal if and only if their domains and codomains are equal. These rules should be very familiar from the the usual rules for simply typed λ-calculus, but are now unsound.

We need only β-reduction, but we need both a dynamically scoped form, which we call β_{Dyn}, and a lexically scoped form, which we call β_{Lex}. Both reduction rules are the usual substitution rules, but β_{Dyn} does substitution that captures free variables, whereas β_{Lex} uses the usual capture-free substitution. For value application,

$$\frac{}{\vdash c : \tau}\,\textbf{Const} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,\textbf{Var}$$

$$\frac{\Gamma,\ x{:}\tau\ \vdash\ E : \sigma}{\Gamma \vdash \lambda x{:}\tau.E : \tau \to \sigma}\,\textbf{Abs} \qquad \frac{\Gamma \vdash E_1 : \tau\ \to\ \sigma \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash (E_1 E_2) : \sigma}\,\textbf{App}$$

$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma, x{:}\tau \vdash E_2 : \sigma}{\Gamma \vdash \texttt{let val } x = E_1 \texttt{ in } E_2 : \sigma}\,\textbf{LetVal} \qquad \frac{\Gamma, t{:}\tau \vdash E : \sigma}{\Gamma \vdash \texttt{let type } t = \tau \texttt{in } E : \sigma}\,\textbf{LetType}$$

$$\frac{\Gamma \vdash E_1 : \texttt{int} \quad \Gamma \vdash E_2 : \texttt{int}}{\Gamma \vdash E_1 + E_2 : \texttt{int}}\,\textbf{Plus} \qquad \frac{\Gamma(t) = \tau \quad \Gamma \vdash E{:}t}{\Gamma \vdash E{:}\tau}\,\textbf{Ident}$$

Figure 4.2: Typing rules for PCF$_\tau$

$$\begin{array}{rcl}
\mathcal{E}[\![c]\!] & = & c \\
\mathcal{E}[\![x]\!] & = & x \\
\mathcal{E}[\![\lambda x{:}\tau.E]\!] & = & \lambda x{:}\tau.\mathcal{E}[\![E]\!] \\
\mathcal{E}[\![E_1 E_2]\!] & = & \mathcal{E}[\![E_1]\!]\mathcal{E}[\![E_2]\!] \\
\mathcal{E}[\![\texttt{let val } x{:}\tau = E_1 \texttt{in } E_2]\!] & = & (\lambda x{:}\tau.\mathcal{E}[\![E_2]\!])(\mathcal{E}[\![E_1]\!]) \\
\mathcal{E}[\![\texttt{let type } t\ = \tau \texttt{ in } E]\!] & = & (\Lambda t.\mathcal{E}[\![E]\!])[\tau]
\end{array}$$

Figure 4.3: Translation from PCF$_\tau$ to second-order λ-calculus.

we use β_{Lex}, for type application, we use β_{Dyn}. We are adopting the usual interpretation of lexical scope as α-conversion of bound variables. Note that this language is *not* confluent, as is normal for dynamically scoped languages [Mit96, page 60].

We demonstrate the unsoundness of the type system with the following example. Consider the statically typable (by the above rules) program:

```
let type t = int in
   let val f:  t -> int = λx:t = x + 1 in
         let type t = bool in
               let val z:t = true in f(z)
```

Which translates to:

$(\Lambda t.(\lambda f{:}t \to \texttt{int}.(\Lambda t.(\lambda z{:}t.\,f(z))(\texttt{true}))[\texttt{bool}])(\lambda x{:}t.x + 1))[\texttt{int}]$

Clearly the application `f(z)` is wrong — its body contains `true + 1`, which is a type error. The program passed the type checking rules, because the free variable *t* got captured. However, without any restrictions on the behavior of dynamic linking, this is *exactly* what would happen in the JVM. In fact, JDK 1.0.*x* did not restrict the behavior of dynamic linking, and so this could actually get past the bytecode verifier. JDK 1.1 enforces restrictions on the behavior of dynamic linking to prevent this example from getting past the bytecode verifier.

4.4 PCF$_{\tau,\rho}$

We now present the second kind of type soundness problem found in Java, which can occur in the presence of multiple `ClassLoaders`. We model multiple `ClassLoaders` with first-class environments. To effectively support first-class environments, we also need type dynamic.

4.4.1 PCF$_{\tau,\rho}$ Definition

We define a language, PCF$_{\tau,\rho}$, which is a combination of call-by-value PCF, first-class environments [Jag94], and dynamic types [ACPP91, LM93]. We use *c* to range over constants (`true, false,` $\{$`..., -2, -1, 0, 1, 2, ...`$\}$, and quoted strings), *x* to range over variables, *t* to range over type variables, and *v* to range over values. All values are tagged with their types. We introduce the semantic objects $\mathit{Closure}, \mathit{Env}, \mathit{Dynamic}, \mathit{Binding}_\nu, \mathit{Binding}_\tau$ as values. λ-abstractions reduce to $\mathit{Closure}$s. `Reify`, `bind`$_\nu$, `bind`$_\tau$ all reduce to Envs. `Dynamic` reduces to $\mathit{Dynamic}$. `Lookup`$_\nu$ reduces to a $\mathit{Binding}_\nu$, and `Lookup`$_\tau$ reduces to a $\mathit{Binding}_\tau$.

$$\begin{array}{rcl}
E & ::= & c \mid x \mid \lambda x{:}\tau.E \mid (E_1\ E_2) \mid (E_1,\ E_2) \\
 & \mid & \texttt{let val}\ x{:}\tau = E_1 \texttt{in}\ E_2 \\
 & \mid & \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \\
 & \mid & E_1 = E_2 \mid E_1 + E_2 \mid \texttt{fst}(E) \mid \texttt{snd}(E) \\
 & \mid & \texttt{typecase}\ E_1\ \texttt{of}\ x{:}\tau\ \Rightarrow\ E_2\ \texttt{else}\ E_3 \\
 & \mid & \texttt{dynamic}(E) \mid \texttt{reify} \mid \texttt{lookup}_\tau(E_1, E_2) \\
 & \mid & \texttt{lookup}_\nu(E_1, E_2) \mid \texttt{bind}_\tau(E_1, E_2) \\
 & \mid & \texttt{bind}_\nu(E_1, E_2) \mid \texttt{eval}(E_1, E_2) \\
\tau_c & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{string} \\
\tau, \sigma & ::= & \tau_c \mid \texttt{env} \mid \texttt{dynamic} \\
 & \mid & \tau\ \rightarrow\ \sigma \mid t \mid \tau\ \times\ \sigma \mid \texttt{binding}_\tau \mid \texttt{binding}_\nu \\
Value & ::= & (c,\ \tau_c) \mid ((v_1,\ v_2),\ \tau \times \sigma) \mid (Env,\ \texttt{env}) \mid (Dynamic(v),\ \texttt{dynamic}) \\
 & \mid & (Closure(\eta,\ x,\ E),\ \tau \rightarrow \sigma) \mid (Binding_\nu(x,\ v),\ \texttt{binding}_\nu) \\
 & \mid & (Binding_\tau(t,\ \tau),\ \texttt{binding}_\tau) \\
Program & ::= & E \mid \texttt{type}\ t = \tau;\ Program
\end{array}$$

Figure 4.4: Grammar for PCF$_{\tau,\rho}$.

We choose this unusual set of features because it captures the aspects of the JVM that we are interested in. We use `reify` to capture the current environment and convert it into a value (*i.e.*, it does for environments what `call/cc` does for continuations). `Lookup(s,e)` takes a string `s` and a reified environment `e`, yielding a binding; `bind(b, e)` takes a binding `b` and a reified environment `e`, and (non-destructively) yields a new reified environment. We have two `lookup` and `bind` operators; one for value bindings, and the other for type bindings. There are no other operations on bindings. The `dynamic` type constructor takes a value and a type (the expression must statically be a member of the type) and returns a value of type `dynamic`. The `typecase` operator takes a value of `dynamic` type, and if its stored type is τ, binds x to the value. We have simplified `typecase` from Abadi, *et al.* [ACPP91] by eliminating their pattern variables, which we do not need. We also have an `eval` operator, which takes an expression (as a `string`) and a reified environment, and returns a value of type `dynamic`.

4.4.2 PCF$_{\tau,\rho}$ Operational Semantics

Instead of translating this language into the second-order λ-calculus, we give the operational semantics in Figure 4.6. In addition to the notation used in Figure 4.4, let s denote a unique stamp, and the _ (underscore) be a pattern matching wild card (as in Standard ML).

4.4.3 PCF$_{\tau,\rho}$: The Type System

Again, the type system for this language in Figure 4.4.3 comes from the simply typed λ-calculus, with the addition of monomorphic type identifiers. We use the traditional $\Gamma, x{:}\tau$ notation for augmenting environments to mean that the definition of x overrides any existing binding in Γ. See Section 4.4.5 for the definition of Γ^*.

Two types, τ, σ are considered the same, written $\Gamma \vdash \tau \cong \sigma$, according to the following rules, where = means syntactic identity:

- If $\tau \in \{\texttt{int}, \texttt{bool}, \texttt{string}, \texttt{env}, \texttt{dynamic}, \texttt{binding}_\nu, \texttt{binding}_\tau\}$, then $\tau = \sigma$.
- If $\tau = \tau_1 \rightarrow \tau_2$, then $\sigma = \sigma_1 \rightarrow \sigma_2$ with $\tau_1 \cong \sigma_1$ and $\tau_2 \cong \sigma_2$.
- If $\tau = \tau_1 \times \tau_2$, then $\sigma = \sigma_1 \times \sigma_2$ with $\tau_1 \cong \sigma_1$ and $\tau_2 \cong \sigma_2$.
- If $\tau = t$, then $\sigma = t$ or $\Gamma(t) \cong \sigma$.
- If $\tau \cong \sigma$, then $\sigma \cong \tau$, *i.e.*, $\cong$ is symmetric.

We define a *semantic entailment* relation, $v \models \tau$, following Leroy and Mauny [LM93], by induction on the structure of τ:

- $\forall \tau . \exists \texttt{error}.\ \texttt{error} \models \tau$
- $c \models \tau_c$ if and only if $c \in \tau_c$
- $(v_1, v_2) \models \tau$ if and only if $\tau = \tau_1 \times \tau_2$ and $v_1 \models \tau_1$ and $v_2 \models \tau_2$.
- $\mathit{Closure}(\eta, x, E) \models \tau$ if and only if $\tau = \tau_1 \rightarrow \tau_2$ and $\exists \Gamma$ such that $\eta \models \Gamma$ and $\Gamma, x{:}\tau_1 \vdash E{:}\tau_2$.
- $\mathit{Binding}_\tau \models \texttt{binding}_\tau$
- $\mathit{Binding}_\nu \models \texttt{binding}_\nu$
- $\mathit{Env} \models \texttt{env}$

We extend the $\models$ relation to environments in a pointwise fashion: $\eta \models \Gamma$ if and only if $\forall x \in \eta . \eta(x) \models \Gamma(x)$.

Define $\hat{\tau}$ to be the set of all type identifiers that appear in a type as type identifiers are replaced with their definition, until the type consists of base types, function types, and pairs.

A type identifier t is *shared* across dynamic environments η_1 and η_2 if and only if: If $\eta_1(t) = (\tau, s)$, then $\eta_2(t) = (\sigma, s)$ and all type identifiers in $\hat{\tau}$ are shared between η_1 and η_2. That is, the definition of t has the same stamp in both environments. Necessarily, $\tau \cong \sigma$ follows, because the two stamps are the same exactly when η_1 and η_2 share the same binding, *i.e.*, the first class environment operators `lookup` and `bind` were used to transport the binding across environments. Note that this sharing relationship is a dynamic attribute of types; the static typing rules only consider a single environment. Our semantics are carefully crafted to avoid name clashes that would compromise type soundness.

We extend this notion of sharing between to type identifiers to a type identifier, t, and a type, τ, defined in η_1 in the obvious fashion: the type identifier is shared with all type identifiers in $\hat{\tau}$ with respect to η_1. Similarly, the notion of sharing between a type, τ with respect to η_1 and an environment, η_2, is defined as all type identifiers appearing in $\hat{\tau}$ with respect to η_1 being shared with all type identifiers bound in η_2.

$$\frac{}{\vdash c : \tau}\ \textbf{Const} \qquad \frac{}{\vdash \texttt{reify} : \texttt{env}}\ \textbf{Reify}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textbf{Var} \qquad \frac{\Gamma,\ x{:}\tau\ \vdash\ E : \sigma}{\Gamma \vdash \lambda x{:}\tau.E : \tau \to \sigma}\ \textbf{Abs}$$

$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma, x{:}\tau \vdash E_2 : \sigma}{\Gamma \vdash \texttt{let val}\ x{:}\tau = E_1\ \texttt{in}\ E_2 : \sigma}\ \textbf{LetVal} \qquad \frac{\Gamma \vdash E_1 : \tau \to \sigma \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash (E_1 E_2) : \sigma}\ \textbf{App}$$

$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1, E_2) : \tau \times \sigma}\ \textbf{Pair} \qquad \frac{\Gamma \vdash E : \sigma \times \tau}{\texttt{fst}(E) : \sigma}\ \textbf{Fst}$$

$$\frac{\Gamma \vdash E : \sigma \times \tau}{\texttt{snd}(E) : \tau}\ \textbf{Snd} \qquad \frac{\Gamma \vdash E_1 : \texttt{bool} \quad \Gamma \vdash E_2 : \tau \quad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 : \tau}\ \textbf{Cond}$$

$$\frac{\Gamma \vdash E_1 : \texttt{int} \quad \Gamma \vdash E_2 : \texttt{int}}{\Gamma \vdash E_1 = E_2 : \texttt{bool}}\ \textbf{Equals} \qquad \frac{\Gamma \vdash E_1 : \texttt{int} \quad \Gamma \vdash E_2 : \texttt{int}}{\Gamma \vdash E_1 + E_2 : \texttt{int}}\ \textbf{Plus}$$

$$\frac{\Gamma \vdash E_1 : \texttt{string} \quad \Gamma \vdash E_2 : \texttt{env}}{\Gamma \vdash \texttt{eval}(E_1, E_2) : \texttt{dynamic}}\ \textbf{Eval} \qquad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{dynamic}(E) : \texttt{dynamic}}\ \textbf{Dynamic}$$

$$\frac{\Gamma \vdash E_1 : \texttt{binding}_\nu \quad \Gamma \vdash E_2 : \texttt{env}}{\Gamma \vdash \texttt{bind}_\nu(E_1, E_2) : \texttt{env}}\ \textbf{Bind}_\nu \qquad \frac{\Gamma \vdash E_1 : \texttt{binding}_\tau \quad \Gamma \vdash E_2 : \texttt{env}}{\Gamma \vdash \texttt{bind}_\tau(E_1, E_2) : \texttt{env}}\ \textbf{Bind}_\tau$$

$$\frac{\Gamma \vdash E_1 : \texttt{string} \quad \Gamma \vdash E_2 : \texttt{env}}{\Gamma \vdash \texttt{lookup}_\nu : \texttt{binding}_\nu}\ \textbf{Lookup}_\nu \qquad \frac{\Gamma \vdash E_1 : \texttt{string} \quad \Gamma \vdash E_2 : \texttt{env}}{\Gamma \vdash \texttt{lookup}_\tau : \texttt{binding}_\tau}\ \textbf{Lookup}_\tau$$

$$\frac{\Gamma \vdash E_1 : \texttt{dynamic} \quad \Gamma, x{:}\tau \vdash E_2 : \sigma \quad \Gamma \vdash E_3 : \sigma}{\Gamma \vdash \texttt{typecase}\ E_1\ \texttt{of}\ x{:}\tau \Rightarrow E_2\ \texttt{else}\ E_3 : \sigma}\ \textbf{TypeCase}$$

$$\frac{t \notin \Gamma^* \quad t \notin \Gamma.\texttt{res}_\tau \quad \Gamma,\ t{:}\tau \vdash \mathit{Program}{:}\sigma}{\Gamma \vdash \texttt{type}\ t = \tau;\ \mathit{Program}{:}\sigma}\ \textbf{Type}$$

Figure 4.5: Typing rules for PCF$_{\tau,\rho}$. Adapted from [LM93, ACPP91]

.

4.4.4 PCF$_{\tau,\rho}$ Domains

In PCF$_{\tau,\rho}$, programs can have both static and checked dynamic errors. Static errors are represented by the special value, `wrong`, of type `Wrong`. Dynamic errors, on the other hand, are represented by the special value `error` of the contextually appropriate type; each type has an `error` value. The semantics are strict with respect to `error`: once a subexpression evaluates to `error`, the result of the entire expression is `error`. The dynamic errors are raised by the environment manipulation operations and `eval`.

4.4.5 PCF$_{\tau,\rho}$ Environments

The key to PCF$_{\tau,\rho}$ is the structure of its environments. We have both static environments, denoted by Γ, and dynamic environments, denoted by η. Static environments are of the form (ρ_ν, ρ_τ, `parent`); type and value bindings are maintained separately. The environment stores a reference to its parent environment, *i.e.*, the environment that was reified to produce this environment. The parent field may be the special value `NULL` to indicate that this is the initial environment. Each dynamic environment is of the form (ρ_ν, ρ_τ, `self`, `parent`, `res`$_\tau$, `res`$_\nu$). That is, name- and type-environments, a unique identifier, the unique identifier of the environment this one was reified from, a set of type identifiers bound in environments derived from this one, and a set of variables bound in environments derived from this one. Identifiers in `res`$_\tau$ cannot be bound in this environment, because they have already been bound in an environment that is a child of this environment, which could produce a name clash between type identifiers. We could relax this rule by using a sharing constraint instead, but the system is already complicated enough. Identifiers in `res`$_\nu$ can not be inserted into this environment via `bind`$_\nu$, to preserve the consistent extension property. We write $\Gamma(x)$ as a shorthand for $\Gamma.\rho_\nu(x)$, and $\Gamma(t)$ for $\Gamma.\rho_\tau(t)$, and similarly for $\eta(x)$ and $\eta(t)$. We write $\emptyset$ for the initial environment, which has no bindings in its ρ_ν and ρ_τ fields, a `NULL parent` field, and no identifiers in its $\texttt{res}_\tau$ field. We use the traditional $\eta, x{:}\tau$ notation for augmenting environments to mean that a new environment is produced where the definition of x overrides any existing binding in $\eta.\rho_\nu$, with the $\eta.\rho_\tau, \texttt{self}, \texttt{parent}, \texttt{res}_\tau$ fields carrying over unchanged from η.

We write Γ^* for the reflexive, transitive closure of Γ with respect to the `parent` field of Γ. So Γ^* is the set of environments on the path from the initial environment to Γ. We define η^* analogously.

We define the notion of *consistent extension* for $\text{PCF}_{\tau,\rho}$ as follows: $\Gamma \preceq \Gamma'$ if and only if:

- $\forall x \in \Gamma.\Gamma(x) = (v, \tau) \Rightarrow \Gamma'(x) = (v', \tau)$.
- $\forall t \in \Gamma.\Gamma(t) = (\tau, s) \Rightarrow \Gamma'(t) = (\tau, s)$.

The corresponding definitions also apply to dynamic environments and reified environments.

In order to support `eval` in an environment built up with `bind` operations, we need complete runtime type information. This defeats the erasure property. However, as a result of static typing, we do *not* have to check types dynamically, except for $\texttt{bind}_\nu$ and `typecase`. We do have to produce types for each value, though.

For brevity, we use two abbreviations. Read $\dfrac{\eta \vdash E_i \rightsquigarrow v_i \quad v_i \not\models \mathtt{int} \quad {\scriptstyle i\,\in 1,2}}{\eta \vdash E_1 + E_2 \rightsquigarrow (\mathtt{wrong}, \mathtt{Wrong})}$ as "If E_1 reduces to v_1 and $v_1 \not\models \mathtt{int}$ or E_2 reduces to v_2 and $v_2 \not\models \mathtt{int}$, then $\eta \vdash E_1 + E_2 \rightsquigarrow (\mathtt{wrong}, \mathtt{Wrong})$.

Read $\dfrac{\eta \vdash E \rightsquigarrow \mathtt{wrong}[]\mathtt{error}}{\eta \vdash \mathtt{fst}(E) \rightsquigarrow (\mathtt{wrong}[]\mathtt{error}, \mathtt{Wrong}[]\alpha)}$ as "If E reduces to wrong or error, then fst(E) reduces to wrong or error, respectively." We use α to stand for the appropriate type for error.

$$\frac{}{\eta \vdash c \rightsquigarrow (c,\, \tau_c,\, \eta.\rho_\tau)}\ \textbf{(1)}$$

$$\frac{(x,\, \tau) \in \eta}{\eta \vdash x \rightsquigarrow (\eta(x),\, \tau,\, \eta.\rho_\tau)}\ \textbf{(2)}$$

$$\frac{}{\eta \vdash \lambda x{:}\tau.E \rightsquigarrow (\mathit{Closure}(\eta, x, E),\, \tau \rightarrow \sigma,\, \eta.\rho_\tau)}\ \textbf{(3)}$$

$$\frac{}{\begin{array}{c}\eta \vdash \mathtt{reify} \rightsquigarrow \\ (\mathit{Env}(\eta.\rho_\nu,\, \eta.\rho_\tau,\, \mathtt{self} \leftarrow s,\, \mathtt{parent} \leftarrow \eta.\mathtt{self},\, \mathtt{res}_\tau \leftarrow \emptyset,\, \mathtt{res}_\nu \leftarrow \emptyset),\, \mathtt{env},\, \eta.\rho_\tau)\end{array}}\ \textbf{(4)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2,\, \_,\, \_) \quad v_1, v_2 \models \mathtt{int}}{\eta \vdash E_1 + E_2 \rightsquigarrow (v_1 + v_2,\, \mathtt{int},\, \eta.\rho_\tau)}\ \textbf{(5)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad v_1 \models \mathtt{bool} \quad v_1 = \mathtt{true} \quad \eta \vdash E_2 \rightsquigarrow (v,\, \tau,\, \_)}{\eta \vdash \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3 \rightsquigarrow (v,\, \tau,\, \eta.\rho_\tau)}\ \textbf{(6)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad v_1 \models \mathtt{bool} \quad v_1 = \mathtt{false} \quad \eta \vdash E_3 \rightsquigarrow (v, \tau,\, \_)}{\eta \vdash \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3 \rightsquigarrow (v,\, \tau,\, \eta.\rho_\tau)}\ \textbf{(7)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1, \tau,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2, \sigma,\, \_)}{\eta \vdash (E_1,\, E_2) \rightsquigarrow ((v_1, v_2),\, \tau \times \sigma,\, \eta.\rho_\tau)}\ \textbf{(8)}$$

$$\frac{\eta \vdash E \rightsquigarrow ((v_1, v_2),\, \tau \times \sigma,\, \_)}{\eta \vdash \mathtt{fst}(E) \rightsquigarrow (v_1,\, \tau,\, \eta.\rho_\tau)}\ \textbf{(9)}$$

$$\frac{\eta \vdash E \rightsquigarrow ((v_1, v_2),\, \tau \times \sigma,\, \_)}{\eta \vdash \mathtt{snd}(E) \rightsquigarrow (v_2,\, \sigma,\, \eta.\rho_\tau)}\ \textbf{(10)}$$

Figure 4.6: Operational Semantics for PCF$_{\tau,\rho}$

$$\frac{\eta \vdash E_1 \rightsquigarrow v_1 \quad \eta, x{:}(v_1,\, \tau,\, \_) \vdash E_2 \rightsquigarrow (v_2,\, \sigma,\, \_)}{\texttt{let val } x{:}\tau = E_1 \texttt{ in } E_2 \;\rightsquigarrow (v_2,\, \sigma,\, \eta.\rho_\tau)} \; \textbf{(11)}$$

$$\frac{\begin{array}{c}\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2,\, \_,\, \_) \\ \forall \gamma \in \eta^* \setminus \eta.\gamma.\texttt{res}_\nu \leftarrow \gamma.\texttt{res}_\nu \cup \{x\} \\ v_1 = \mathit{Binding}_\nu(x,\, v,\, \eta) \models \texttt{binding}_\nu \quad v_2 \models \texttt{env} \quad \mathit{Ok}_\nu(v_1, v_2)\end{array}}{\eta \vdash \texttt{bind}_\nu(E_1, E_2) \rightsquigarrow (\mathit{Env}(v_1, v_2), \texttt{env},\, \eta.\rho_\tau)} \; \textbf{(12)}$$

$$\frac{\begin{array}{c}\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2,\, \_,\, \_) \\ \forall \gamma \in \eta^* \setminus \eta.\gamma.\texttt{res}_\tau \leftarrow \gamma.\texttt{res}_\tau \cup \{t\} \\ v_1 = \mathit{Binding}_\tau(t,\, \tau,\, \eta) \models \texttt{binding}_\tau \quad v_2 \models \texttt{env} \quad \mathit{Ok}_\tau(v_1, v_2)\end{array}}{\eta \vdash \texttt{bind}_\tau(E_1, E_2) \rightsquigarrow (\mathit{Env}(v_1, v_2), \texttt{env},\, \eta.\rho_\tau)} \; \textbf{(13)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2,\, \_,\, \_) \quad v_1 \models \texttt{string} \quad v_2 \models \texttt{env}}{\eta \vdash \texttt{lookup}_\nu(E_1, E_2) \rightsquigarrow (\mathit{Binding}_\nu(x, v, \eta),\, \texttt{binding}_\nu,\, \eta.\rho_\tau)} \; \textbf{(14)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2,\, \_,\, \_) \quad v_1, v_2 \models \texttt{int} \quad v_1 = v_2}{\eta \vdash E_1 = E_2 \rightsquigarrow (\texttt{true},\, \texttt{bool},\, \eta.\rho_\tau)} \; \textbf{(15)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow v_1 \quad \eta \vdash E_2 \rightsquigarrow v_2 \quad v_1, v_2 \models \texttt{int} \quad v_1 \neq v_2}{\eta \vdash E_1 = E_2 \rightsquigarrow (\texttt{false},\, \texttt{bool})} \; \textbf{(16)}$$

$$\frac{\begin{array}{c}\eta \vdash E_1 \rightsquigarrow \mathit{Dynamic}(v_1,\, \tau, \eta'.\rho_\tau) \quad \eta, x{:}(v_1, \tau) \vdash E_2 \rightsquigarrow (v_2, \sigma) \\ \mathit{Shared}((\tau, \eta'.\rho_\tau), \eta)\end{array}}{\texttt{typecase } E_1 \texttt{ of } x{:}\tau \Rightarrow E_2 \texttt{ else } E_3 \rightsquigarrow (v_2,\, \sigma,\, \eta.\rho_\tau)} \; \textbf{(17)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow \mathit{Dynamic}(v_1,\, \sigma,\, \_) \quad \eta \vdash E_3 \rightsquigarrow (v_2,\, \sigma,\, \_)}{\texttt{typecase } E_1 \texttt{ of } x{:}\tau \Rightarrow E_2 \texttt{ else } E_3 \rightsquigarrow (v_2,\, \sigma,\, \eta.\rho_\tau)} \; \textbf{(18)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow (v_1,\, \_,\, \_) \quad \eta \vdash E_2 \rightsquigarrow (v_2,\, \_,\, \_) \quad v_1 \models \texttt{string} \quad v_2 \models \texttt{env}}{\eta \vdash \texttt{lookup}_\tau(E_1, E_2) \rightsquigarrow (\mathit{Binding}_\tau(t, \tau, \eta), \texttt{binding}_\tau,\, \eta.\rho_\tau)} \; \textbf{(19)}$$

$$\frac{\eta \vdash E \rightsquigarrow (v,\, \tau,\, \_)}{\texttt{dynamic}(E) \rightsquigarrow (\mathit{Dynamic}(v,\, \tau,\, \eta.\rho_\tau),\, \texttt{dynamic},\, \eta.\rho_\tau)} \; \textbf{(20)}$$

$$\frac{\begin{array}{c}\eta \vdash E_1 \rightsquigarrow (\mathit{Closure}(\eta', x, E'), \tau \rightarrow \sigma,\, \_) \\ \eta \vdash E_2 \rightsquigarrow (v_1, \tau,\, \_) \quad \eta', x{:}(v_1, \tau) \vdash E' \rightsquigarrow (v_2, \sigma,\, \_)\end{array}}{\eta \vdash (E_1 E_2) \rightsquigarrow (v_2,\, \sigma,\, \eta.\rho_\tau)} \; \textbf{(21)}$$

$$\frac{\eta \vdash E_1 \leadsto (v_1,\,\_,\,\_) \quad v_1 \models \texttt{string} \quad \eta \vdash E_2 \leadsto (v_2,\,\_,\,\_) \quad v_2 \models \texttt{env} \qquad \exists \tau . v_2 \vdash v_1{:}\tau \quad v_2 \vdash v_1 \leadsto (v,\,\sigma,\,\eta'.\rho_\tau)}{\eta \vdash \texttt{eval}(E_1, E_2) \leadsto (\mathit{Dynamic}(v,\,\sigma,\,\eta'.\rho_\tau), \texttt{dynamic}),\,\eta.\rho_\tau} \quad \textbf{(22)}$$

$$\frac{\forall \gamma \in \eta^* \setminus \eta . \gamma.\texttt{res}_\tau \leftarrow \gamma.\texttt{res}_\tau \cup t \qquad \eta,\, t{:}\tau \vdash \mathit{Program} \leadsto (v, \tau,\, \eta'.\rho_\tau)}{\eta \vdash \texttt{type}\ t = \tau;\ \mathit{Program} \leadsto (v, \tau,\, \eta'.\rho_\tau)} \quad \textbf{(23)}$$

$$\frac{\eta \vdash E \leadsto \texttt{wrong[]error}}{\eta \vdash \texttt{fst}(E) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(24)}$$

$$\frac{\eta \vdash E \leadsto \texttt{wrong[]error}}{\eta \vdash \texttt{snd}(E) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(25)}$$

$$\frac{\eta \vdash E_i \leadsto \texttt{wrong[]error} \quad i \in 1{,}2}{\eta \vdash E_1 = E_2 \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(26)}$$

$$\frac{\eta \vdash E_1 \leadsto \texttt{wrong[]error}}{\eta \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(27)}$$

$$\frac{\eta \vdash E_1 \leadsto \texttt{true} \quad \eta \vdash E_2 \leadsto \texttt{wrong[]error}}{\eta \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(28)}$$

$$\frac{\eta \vdash E_1 \leadsto \texttt{false} \quad \eta \vdash E_3 \leadsto \texttt{wrong[]error}}{\eta \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(29)}$$

$$\frac{\eta \vdash E_i \leadsto \texttt{wrong[]error} \quad i \in 1{,}2}{\eta \vdash \texttt{bind}_\tau(E_1, E_2) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(30)}$$

$$\frac{\eta \vdash E_1 \leadsto v \quad v \neq \mathit{Closure}(\eta', x, E')}{\eta \vdash (E_1 E_2) \leadsto (\texttt{wrong}, \texttt{Wrong},\, \eta.\rho_\tau)} \quad \textbf{(31)}$$

$$\frac{\eta \vdash E_i \leadsto \texttt{wrong[]error} \quad i \in 1{,}2}{\eta \vdash (E_1 E_2) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha,\, \eta.\rho_\tau)} \quad \textbf{(32)}$$

$$\frac{\eta \vdash E_1 \leadsto v_1 \quad \eta \vdash E_2 \leadsto v_2 \quad v_1 \models \texttt{binding}_\nu \quad v_2 \models \texttt{env} \quad \neg Ok_\nu(v_1, v_2)}{\eta \vdash \texttt{bind}_\nu(E_1, E_2) \leadsto (\texttt{error}, \texttt{env},\, \eta.\rho_\tau)} \quad \textbf{(33)}$$

$$\frac{\eta \vdash E_1 \leadsto v_1 \quad \eta \vdash E_2 \leadsto v_2 \quad v_1 \models \texttt{binding}_\tau \quad v_2 \models \texttt{env} \quad \neg Ok_\tau(v_1, v_2)}{\eta \vdash \texttt{bind}_\tau(E_1, E_2) \leadsto (\texttt{error}, \texttt{env},\, \eta.\rho_\tau)} \quad \textbf{(34)}$$

$$\frac{\eta \vdash E \leadsto v_i \quad v_i \not\models \texttt{int} \quad {\scriptstyle i \in 1,2}}{\eta \vdash E_1 + E_2 \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{35}$$

$$\frac{\eta \vdash E_1 \leadsto v \quad v \not\models \texttt{binding}_\tau}{\eta \vdash \texttt{bind}_\tau(E_1, E_2) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{36}$$

$$\frac{\eta \vdash E_2 \leadsto v \quad v \not\models \texttt{env}}{\eta \vdash \texttt{bind}_\tau(E_1, E_2) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{37}$$

$$\frac{\eta \vdash E \leadsto \texttt{wrong[]error}}{\texttt{dynamic}(E) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha, \eta.\rho_\tau)} \tag{38}$$

$$\frac{\eta \vdash E_1 \leadsto \texttt{dynamic}(v_1, \tau, \eta) \quad \eta, x{:}(v_1, \tau) \vdash E_2 \leadsto \texttt{wrong[]error}}{\texttt{typecase } E_1 \texttt{ of } x{:}\tau \Rightarrow E_2 \texttt{ else } E_3 \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha, \eta.\rho_\tau)} \tag{39}$$

$$\frac{\eta \vdash E_1 \leadsto v \quad v \not\models \texttt{string}}{\eta \vdash \texttt{lookup}_\nu(E_1, E_2) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{40}$$

$$\frac{\eta \vdash E_2 \leadsto v \quad v \not\models \texttt{env}}{\eta \vdash \texttt{lookup}_\nu(E_1, E_2) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{41}$$

$$\frac{\eta \vdash E_i \leadsto \texttt{wrong[]error} \quad {\scriptstyle i \in 1,2}}{\eta \vdash \texttt{lookup}_\tau(E_1, E_2) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha, \eta.\rho_\tau)} \tag{42}$$

$$\frac{\eta \vdash E_i \leadsto \texttt{wrong[]error} \quad {\scriptstyle i \in 1,2}}{\eta \vdash \texttt{lookup}_\nu(E_1, E_2) \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha, \eta.\rho_\tau)} \tag{43}$$

$$\frac{\eta \vdash E_1 \leadsto v \quad v \not\models \texttt{string}}{\eta \vdash \texttt{lookup}_\tau(E_1, E_2) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{44}$$

$$\frac{\eta \vdash E_2 \leadsto v \not\models \texttt{env}}{\eta \vdash \texttt{lookup}_\tau(E_1, E_2) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{45}$$

$$\frac{\eta \vdash E_1 \leadsto \texttt{dynamic}(v_1, \sigma) \quad \eta \vdash E_3 \leadsto \texttt{wrong[]error}}{\texttt{typecase } E_1 \texttt{ of } x{:}\tau \Rightarrow E_2 \texttt{ else } E_3 \leadsto (\texttt{wrong[]error}, \texttt{Wrong[]}\alpha, \eta.\rho_\tau)} \tag{46}$$

$$\frac{\eta \vdash E \leadsto v \quad v \neq (v_1, v_2)}{\eta \vdash \texttt{fst}(E) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{47}$$

$$\frac{\eta \vdash E \leadsto v \quad v \neq (v_1, v_2)}{\eta \vdash \texttt{snd}(E) \leadsto (\texttt{wrong}, \texttt{Wrong}, \eta.\rho_\tau)} \tag{48}$$

$$\frac{\eta \vdash E_i \rightsquigarrow \texttt{wrong}[]\texttt{error} \qquad i \in 1,2}{\texttt{let val } x{:}\tau = E_1 \texttt{ in } E_2 \rightsquigarrow (\texttt{wrong}[]\texttt{error},\, \texttt{Wrong}[]\alpha,\, \eta.\rho_\tau)} \; \textbf{(49)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow v_1 \qquad v_1 \not\models \texttt{string}}{\eta \vdash \texttt{eval}(E_1, E_2) \rightsquigarrow (\texttt{wrong}, \texttt{Wrong},\, \eta.\rho_\tau)} \; \textbf{(50)}$$

$$\frac{\eta \vdash E_2 \rightsquigarrow v_2 \qquad v_2 \not\models \texttt{env}}{\eta \vdash \texttt{eval}(E_1, E_2) \rightsquigarrow (\texttt{wrong}, \texttt{Wrong},\, \eta.\rho_\tau)} \; \textbf{(51)}$$

$$\frac{\eta \vdash E_i \rightsquigarrow \texttt{wrong}[]\texttt{error} \qquad i \in 1,2}{\eta \vdash \texttt{eval}(E_1, E_2) \rightsquigarrow (\texttt{wrong}[]\texttt{error}, \texttt{Wrong}[]\alpha,\, \eta.\rho_\tau)} \; \textbf{(52)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow v \qquad v \not\models \texttt{binding}_\nu}{\eta \vdash \texttt{bind}_\nu(E_1, E_2) \rightsquigarrow (\texttt{wrong},\, \texttt{Wrong},\, \eta.\rho_\tau)} \; \textbf{(53)}$$

$$\frac{\eta \vdash E_2 \rightsquigarrow v \not\models \texttt{env}}{\eta \vdash \texttt{bind}_\nu(E_1, E_2) \rightsquigarrow (\texttt{wrong},\, \texttt{Wrong},\, \eta.\rho_\tau)} \; \textbf{(54)}$$

$$\frac{\eta \vdash E_i \rightsquigarrow \texttt{wrong}[]\texttt{error} \qquad i \in 1,2}{\eta \vdash \texttt{bind}_\nu(E_1, E_2) \rightsquigarrow (\texttt{wrong}[]\texttt{error},\, \texttt{Wrong}[]\alpha,\, \eta.\rho_\tau)} \; \textbf{(55)}$$

$$\frac{\eta \vdash E_1 \rightsquigarrow \texttt{wrong}[]\texttt{error}}{\texttt{typecase } E_1 \texttt{ of } x{:}\tau \Rightarrow E_2 \texttt{ else } E_3 \rightsquigarrow (\texttt{wrong}[]\texttt{error},\, \texttt{Wrong}[]\alpha,\, \eta.\rho_\tau)} \; \textbf{(56)}$$

where

$$\begin{array}{rcl}
Ok_\nu(\eta, Binding_\nu(x, (v, \tau, \eta'))) & \iff & \text{if } x \in \eta \text{ then } \eta(x) = (v', \tau) \text{ otherwise true} \\
& \wedge & Shared(\eta, (\tau, \eta'.\rho_\tau)) \\
Ok_\tau(\eta, Binding_\tau(t, \sigma, \eta')) & \iff & \neg(t \in \eta^*) \wedge\ Shared(\eta, (\sigma, \eta'.\rho_\tau)) \\
& \wedge & t \notin \eta.\texttt{res}_\tau \\
Shared(\eta, (\tau, \eta'.\rho_\tau)) & \iff & \forall t \in \hat{\tau}.Shared(\eta.\rho_\tau,\, t,\, \eta'.\rho_\tau) \\
Shared(\eta.\rho_\tau,\, t,\, \eta'.\rho_\tau) & \iff & \eta(t) = (\_, s) \Rightarrow \eta'(t) = (\_, s)
\end{array}$$

4.4.6 PCF$_{\tau,\rho}$ Example

Consider the code in figure 4.7.

```
let val e1:env = typecase (eval("type u = int ; type t = u * int ;
                                let val f:  t -> int =
                                 λx:t.fst(x) + 1
                                in reify",
                               reify))
              of y:env => y else reify in
      let val b1:binding_τ = lookup_τ("t", e1) in
      let val b2:binding_ν = lookup_ν("f", e1) in
      let val e2:env = bind_ν(b2, (bind_τ(b1, reify))) in
      eval("type u = bool ; let val y:u = true in
      let val z:t = (y,3) in f(z)", e2)
```

Figure 4.7: Example program in PCF$_{\tau,\rho}$ syntax showing free variable capture across environments. This program reduces to `error` in PCF$_{\tau,\rho}$.

Although this program may seem complicated, it is actually quite simple. We will consider how the program would evaluate *without* enforcement of the sharing constraints, *i.e.*, if the Shared hypotheses were removed from the semantics. The first `let` binds `e1` to the environment produced by evaluating the text in quotes. Next, we lookup the type-binding of `t` (*i.e.*, `u` $\times$ `int`), into the variable `b1`. Similarly, we set `b2` to the binding of f in `e1`, *i.e.*, $\lambda x{:}t.\mathit{fst}(x)+1$. Next we construct a new environment, `e2`, which is the current environment, plus the bindings of f and t from `e1`. Now, we evaluate the declaration `type u = bool`, which binds the free variable u in t (imported from `e1`) in `e2` resulting in e_2'. Finally, in e_2', we evaluate `let val y:u = true in let val z:t = (y,3) in f(z)`. Reducing the `let` expressions finally yields `f(true, 3)`, which reduces to $(\lambda x{:}u.x+1)(\mathit{fst}(\mathit{true} : u, 3 : \mathit{int}))$, which reduces to `true + 1`. Again, we have a type-soundness failure. Our semantics for *PCF*$_{\tau,\rho}$ reduces this program to `error`, because the Shared hypotheses fails when we attempt to construct `e2`.

The problem we see here is extracting a binding from one environment, and placing it another environment. Again, this is just another instance of free variable capture that leads to an unsound typing judgement; see Figure 4.8.

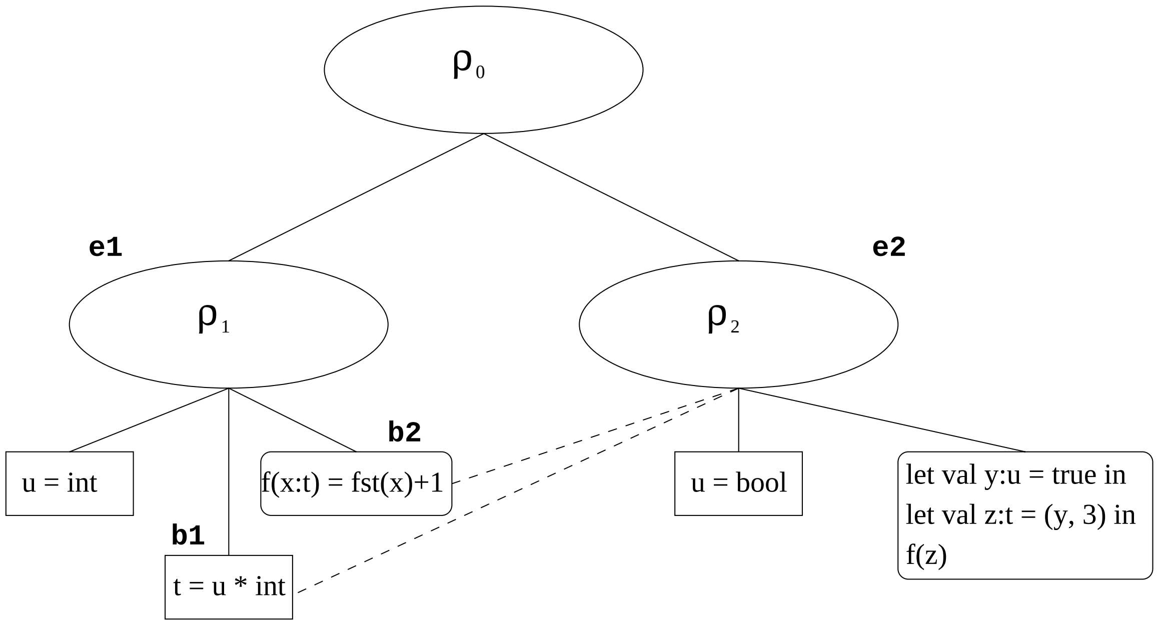

Figure 4.8: This figure shows a cross-environment lookup, which would result in a type-soundness failure as the program attempts to evaluate `true + 1`. Our semantics yields the result `error`. The labels correspond to the variables in the code in Figure 4.7.

4.5 Solution for PCF$_{\tau,\rho}$

Recall that lexical scope can be explained via α-conversion. The only difference in the operational semantics is that bound variables get renamed under a lexically scoped system to avoid the name clashes that plague dynamic scope. So far, Java systems have tried to avoid this problem via *ad hoc* restrictions on dynamic linking. JDK 1.1 uses a hash table to ensure that a class is never redefined in a given environment, as I suggested in Chapter 3, after my discovery of this problem in

JDK 1.0. However, this misses the name capture in the second example, above.

With our new understanding of the problem, we can justify a sufficient set of restrictions on environment manipulation (*i.e.*, dynamic linking) to ensure soundness of the type system. Note that this is not the only possible set of restrictions.

We have three restrictions:

Restriction 4.1 *A type identifier may never be defined more than once in a type environment.*

Restriction 4.2 *Each reified environment must be a* consistent extension *of its parent;* i.e., *if a name is defined in both environments, they must be bound to the same thing.*

Restriction 4.3 *If an environment references a binding originally defined in a different environment, the two environments must share all the type bindings referenced (directly or indirectly) by the shared binding.*

Note Restriction 4.3 does not subsume restriction 4.2, because we insist that all environments be consistent extensions of their parent environment, even if they do not refer to any symbols in their parent environment.

We examine the program in Figure 4.7 again in light of these restrictions. The program will fail, *i.e.*, evaluate to `error`, when it tries to evaluate `bind`$_\tau$`(b1, reify)` because it runs afoul of restriction 3: environments ρ_1 and ρ_2 do not share a binding for `u`.

4.5.1 Formalization

We now formalize the above intuitions. The typing rules in Figure 4.4.3 and operational semantics in Figure 4.6 are adapted from Leroy and Mauny [LM93], Jagannathan [Jag94] and Abadi, *et al.* [ACPP91]. Because our system is monomorphic, the discussion of open *vs.* closed dynamic types does not arise. We allow free type identifiers to appear in dynamic types. Our system has a minor twist, also exhibited by Java: types are name-equivalent statically, and (name, environment)-equivalent dynamically. That is, `typecase` requires any type identifiers in the pattern be bound to the same definitions in the environments where the dynamic value is created and used.

The formal semantics are in Figure 4.6. We take the slightly unusual approach of expressing evaluation as a relation between environment $\times$ expression $\leadsto$ value $\times$ type $\times$ type environment. We need values to carry their types, so that we can build up environments and subsequently type check expressions in those environments. In order to interpret any type identifiers that might appear in a type, we need a type environment. However, this leads to phase distinction [Car88a] problems: execution is no longer separate from type checking, because execution, by altering the environment, can invalidate a typing judgement. Thus, we put runtime checks on the `bind` operation to ensure the continued validity of the static typing judgement. The environment operations are easy to implement in an interpreted setting, but more difficult in a compiled setting. However, one sees the same general problem in any compiled setting with dynamic linking. The first time a symbol is referenced, it has to be resolved; afterwards it can be directly accessed.

We begin with a technical lemma about the structure of environments in the system.

Lemma 4.1 (Environments form a tree) *The graph formed by the* `parent` *fields of all environments is a tree.*

Proof: By induction on the number of environments. Initially, there is one environ-

ment, and any one node graph is a tree. For the induction step, `reify` is the only operation that sets `parent`, and it always sets it to the current environment.

We present a slight variation on the standard type-soundness proof. Each type contains `error`, a constant used to denote checked runtime errors. We show that a well-typed program never evaluates to `wrong`.

Lemma 4.2 (Consistent Extensions Preserve Types) *If $\Gamma \vdash E{:}\tau$ and $\Gamma \preceq \Gamma'$, then $\Gamma' \vdash E{:}\tau$.*

Proof: By induction on the height of the typing derivation. There is one case for each typing rule; we consider each rule as the last step in a typing derivation.

- *The result follows immediately for **Const** and **Reify**.*
- *For **Var**, $\Gamma \vdash x{:}\tau$ and $\Gamma \preceq \Gamma'$ immediately implies $\Gamma' \vdash x{:}\tau$ by the definition of consistent extension.*
- *Using the induction hypothesis, the result follows immediately for **Pair, Fst, Snd, Cond, Equals, Plus, Eval, Dynamic, Bind$_\nu$, Bind$_\tau$, Lookup$_\nu$, Lookup$_\tau$**, and **App**.*
- *For the binding operators, (**Abs, LetVal**, and **TypeCase**), $\Gamma \preceq \Gamma'$ implies $\Gamma, x{:}\tau \preceq \Gamma', x{:}\tau$, because $\forall y \in \Gamma, \Gamma(y){:}\tau$ implies $\Gamma'(y){:}\tau$ and $\Gamma(x){:}\tau$ and $\Gamma'(x){:}\tau$. By the induction hypothesis, the types of the subterms are preserved, so the result follows.*
- *For type declarations, using **Type**, consistent extensions may cause inadmissibility. However, this is a static error. In the case of `eval`, a static error will cause `error` (of type dynamic) to be returned, which preserves the type of `eval`.*

Lemma 4.3 (Environment Consistency) *Every expression is reduced in a consistent extension of the environment it was type-checked in, and reified environments remain consistent extensions of the environment that was captured. This implies that runtime environments are consistent with type checking environments.*

Proof: Proceeds by induction on the length of the reduction.

- *The result is immediate for values, because either (rule 1) applies, and the result is immediate, or no reduction rules apply.*
- *For variables (rule 2), the result is immediate, because the environment does not change.*
- *None of the reduction rules that reduce to* `wrong` *or* `error` *(rules 24–56) change the environment, so the result follows from the inductive hypothesis in conjunction with the typing rules for each term.*
- *The result follows from the inductive hypothesis, the typing rules, and the reflexivity of consistent extension, for pairing (rule 8), addition (rule 5), equality (rules 15–16),* `if-then-else` *(rules 6–7),* `fst` *(rule 9),* `snd` *(rule 10),* $\texttt{lookup}_{\nu}$ *(rule 14), and* $\texttt{lookup}_{\tau}$ *(rule 19).*
- *For reify (rule 4), $Env(\eta)$ captures the bindings of η, so by reflexivity, the reified environment is a consistent extension of Γ.*
- *For λ-abstraction (rule 3), the closure built contains η, a consistent extension of itself. Evaluation directly leads to a $Closure$, so the result is immediate.*
- *For* `dynamic` *(rule 20), the value built contains η, a consistent extension of itself. Evaluation directly leads to a $Dynamic$, so the result is immediate.*

- *For function application (rule 21), the induction hypothesis tells us that consistent extension is preserved as we reduce the subexpressions to a $Closure$, and an argument. Now, we augment the environment stored in the closure with a binding for the function argument, which is exactly what the* ***Abs*** *rule did for type-checking the body of the closure, which is the desired result.*

- *For `typecase` (rules 17–18), the inductive hypothesis allows us to conclude the consistent extension property for the evaluation of the dynamic value whose type is being tested. We proceed with a case split on whether the expression is a member of τ. If it is, we evaluate the body in an environment augmented with a binding for x, just as in the typing rule,* ***TypeCase****. We ensure that any type identifiers are shared, so that no inconsistency arises. Combined with the inductive hypothesis, this leads to the desired result. Otherwise, the `else` branch is evaluated in the same environment as the `typecase`, which also corresponds to the typing rule. Combined with the inductive hypothesis, this leads to the result.*

- *For `let val` (rule 11), the argument follows `typecase`, above, where the expression is a member of the type.*

- *For bind$_\nu$ (rule 12), the induction hypothesis covers consistent extension when reducing the subterms. OK_ν ensures that the variable is either unbound in v$_2$, in which case the consistent extension property with respect to v$_2$ is preserved, or that the variable denoted by v$_1$ is bound to a value of the same type as the value in the binding v$_2$, which also preserves consistent extension.*

- *For bind$_\tau$ (rule 13), the induction hypothesis yields that the reductions leading to v$_1$ and v$_2$ preserve consistent extension. Ok_τ ensures that the variable in v$_1$ is not bound in v$_2$ or in the reflexive, transitive closure of the parent(s) of v$_2$. The Ok_τ predicate* *also ensures that all type variables reachable in the expansion of v_1 are bound to the same definitions in v_2 as in the environment that v_1 was originally defined in. Thus, the new environment is a consistent extension of v_2.*

- *For* `eval` *(rule 22), the inductive hypothesis ensures consistent extension during the evaluation of the two arguments. The first argument, a string, is type checked, with respect to the second argument, an environment. It is then evaluated in the same environment. This is the desired result.*

- *For* `type` $t = \tau$; *Program (rule 23), the declaration is either admissible or not. If t is unbound and $t \notin \eta.\texttt{res}_\tau$, the environment is augmented, and the new environment is a consistent extension by definition. The inductive hypothesis, applied to the remainder of the program yields the desired result.*

Theorem 4.4 (Subject Reduction) *If $\Gamma \vdash$ Program : τ and $\eta \models \Gamma$, and $\eta \vdash$ Program $\rightsquigarrow$ (v, τ, ρ_τ), then $v \models \tau$*

The proof proceeds by induction on the derivation. As before, we can ignore all rules with a hypothesis that reduces to `wrong`*, as these violate the induction hypothesis. All rules which reduce to* `error` *produce an* `error` *value of the appropriate type, so subject reduction holds for them.*

- *The result is immediate for values, from (rule 1) or because no other reduction rules apply to them.*

- *For reducing variables, there is only one reduction rule (rule 2), and the expression must have been typed using* ***Var***. *Using $\eta \models \Gamma$, the result is immediate.*

- *For* `let val x:`$\tau = E_1$ `in` E_2 *(rule 11), the last step of the typing derivation must have used* ***LetVal***. *By the induction hypothesis, $E \models \tau$, so $x{:}\tau$. Either $E_1 \rightsquigarrow$* `error`*,* *in which case E_2 evaluates to* `error` *(of the same type as E_2), or $\eta, x{:}\tau \vdash E_2 \rightsquigarrow v$, and $v \models \sigma$ because $\eta, x{:}(v, \tau) \models \Gamma, x{:}\tau$, at which point the induction hypothesis leads to the desired conclusion.*

- *For* `if-then-else` *(rules 6–7),* ***Cond*** *must have been the last typing rule used. By the induction hypothesis, $E_1 \rightsquigarrow v_1$, and $v_1 \models$* `bool`*, and E_2 or E_3 reduce to v_2, and $v_2 \models \tau$, as desired. The cases for $E = E$ (rules 15–16), and $E + E$ (rule 5) are similar and omitted.*

- *For λ abstractions, we have $\lambda x{:}\tau.E$ which must have been typed used* ***Abs****. But it reduces to $\mathit{Closure}(\eta, x, E)$ (rule 3), and $\mathit{Closure}(\eta, x, E) \models \tau \rightarrow \sigma$, as needed.*

- *For function application, we have $v_1 \rightsquigarrow \mathit{Closure}(\eta', x, E')$, and $v_1 \models \tau \rightarrow \sigma$, from the induction hypothesis. In rule 21, $\eta, x{:}(v, \tau) \models \Gamma, x{:}\tau$, and so $v_2 \models \sigma$ by the induction hypothesis. This is the desired result.*

- *For pairs, the last typing rule used must have been* ***Pair****, so the inductive hypothesis yields $v_1 \models \tau$ and $v_2 \models \sigma$, so (rule 8) $(v_1, v_2) \models \tau \times \sigma$ by the definition of $\models$.*

- *For* `fst` *(rule 9), the induction hypothesis yields $(v_1, v_2) \models \tau \times \sigma$, so $v_1 \models \tau$, by definition of $\models$. The case for* `snd` *(rule 10) is analogous and omitted.*

- *For* `reify` *(rule 4), the expression must have been typed using* ***Reify****, yielding type* `env`*. But the reduction rule yields Env, and $\mathit{Env} \models$* `env` *by definition.*

- *For $\texttt{lookup}_\nu$ (rule 14), the expression must have been typed using* ***Lookup****$_\nu$, and the induction hypothesis yields $v_1 \models$* `string` *and $v_2 \models$* `env`*. The only applicable reduction rule reduces to $\mathit{Binding}_\nu$, and $\mathit{Binding}_\nu \models \texttt{binding}_\nu$.*

- *For $\texttt{lookup}_\tau$ (rule 19), the expression must have been typed using* ***Lookup****$_\tau$, and the induction hypothesis yields $v_1 \models$* `string` *and $v_2 \models$* `env`*. The only applicable* *reduction rule reduces to $\mathit{Binding}_\tau$, and $\mathit{Binding}_\tau \models \texttt{binding}_\tau$.*

- *For $\texttt{bind}_\nu$ (rule 12), the expression must have been typed using **Bind**$_\nu$, and the induction hypothesis yields $v_1 \models \texttt{binding}_\nu$ and $v_2 \models \texttt{env}$. The only applicable reduction rule yields Env, and $\mathit{Env} \models \texttt{env}$, as desired.*

- *For $\texttt{bind}_\tau$ (rule 13), the expression must have been typed using **Bind**$_\tau$, and the induction hypothesis yields $v_1 \models \texttt{binding}_\tau$ and $v_2 \models \texttt{env}$. The only applicable reduction rule yields Env, and $\mathit{Env} \models \texttt{env}$, as desired.*

- *For `typecase` E_1 `of` `x`$:\tau \Rightarrow E_2$ `else` E_3 (rules 17–18), it must have been typed using **TypeCase**. In conjunction with the induction hypothesis, we have $v_1 \models \texttt{dynamic}$. If v_1 is of the form $\mathit{Dynamic}(v_2,\ \tau,\ \rho_\tau)$, and $\mathit{Shared}(\eta, (\tau, \rho_\tau)$), then $\eta, x{:}\tau \vdash E_2 \rightsquigarrow v_3$, and $v_3 \models \sigma$ because $\eta, x{:}(v, \tau) \models \Gamma, x{:}\tau$, at which point the induction hypothesis leads to the desired conclusion. Otherwise, we appeal to the induction hypothesis for $E_3 \rightsquigarrow (v_2, \sigma)$ as needed.*

- *For `dynamic` (rule 20), the last typing rule used must have been **Dynamic**, so $\exists \tau \vdash E{:}\tau$. By the induction hypothesis, $E \rightsquigarrow (v,\ \tau,\ \rho_\tau)$. The only reduction rule yields $\mathit{Dynamic}(v,\ \tau,\ \rho_\tau)$, and $\mathit{Dynamic}(v,\ \tau,\ \rho_\tau) \models \texttt{dynamic}$ by definition.*

- *For `eval` (rule 22), the last typing rule used must have been **Eval**. The induction hypothesis yields $v_1 \models \texttt{string}$ and $v_2 \models \texttt{env}$. If v_1 does not denote a well typed program with respect to v_2, the semantics produce (`error`, `dynamic`) (the `error` value at type `dynamic`), and `error` $\models$ `dynamic`. Otherwise, the induction hypothesis applies to $v_2 \vdash v_1 \rightsquigarrow (v,\ \tau,\ \rho_\tau)$, and `eval` yields $\mathit{Dynamic}(v,\ \tau,\ \rho_\tau)$, and $\mathit{Dynamic}(v,\ \tau,\ \rho_\tau) \models \texttt{dynamic}$.*

- *For `type` $t = \tau$ (rule 23), the declaration must have been typed using **Type**. Since **Type** requires the environment to be well-formed, we cannot go wrong here.*

In the above proofs, we take great advantage of the fact that evaluating an expression does not change the type environment. This allows us to simplify the argument, because whether a type declaration is admissible or not cannot change between type checking and evaluation. Therefore, we only have to ensure that the first class environment operators and `typecase` (because the dynamic value may have been constructed in a different environment via `eval`) have appropriate run-time checks.

Corollary 4.5 (Type Soundness) *If* $\emptyset \vdash$ *Program* $: \tau$ *and Program* $\leadsto (v, \tau, \rho_\tau)$*, then* $v \neq$ `wrong`.

Proof: Note that there is no introduction rule for `Wrong`*. In conjunction with Theorem 4.4, this implies* $v \neq$ `wrong`.

The proof technique used in Corollary 4.5 is a common trick I first saw used in Abadi, *et al* [ACPP91]. Many others have used the same technique.

4.6 Relating PCF$_{\tau,\rho}$ to Java

Now that we have studied the phenomenon of dynamically scoped types in PCF$_{\tau,\rho}$, we need to relate our model back to Java. While the PCF$_{\tau,\rho}$ model may seem complicated, its complexity is needed to capture the appropriate semantics of Java and the JVM. PCF$_{\tau,\rho}$ is still much simpler than the JVM. We now turn to matching PCF$_{\tau,\rho}$'s types, environments, typing rules, and operators to JVM's classes, ClassLoaders, type system, and operators.

We regret that this chapter furthers the confusion between types and classes, but there is no other simple alternative. Note that we do not really care about the definition of a class: all we need to capture is the intuitive notion of a combination of a type declaration and some code.

We have come to one of the deep, dark corners of Java: the ClassLoader. The basic JVM knows how to load bytecode only from the local file system. To load code from anywhere else (*i.e.,* the World Wide Web), the runtime system exports an interface (`defineClass()`) for turning an array of bytes into a `Class` object as a method of `java.lang.ClassLoader`. By subclassing ClassLoader, the developer can use the `defineClass()` method to construct classes from arbitrary byte sources (as long as the bytes are the representation of a valid class in the Java Class file format).

Each class object (*i.e.*, instance of the class `java.lang.Class`) contains a reference to the ClassLoader that defined it. In addition to providing an interface to turn bytes into classes, the ClassLoader also acts as an environment; the runtime makes an upcall to retrieve the definitions of other classes mentioned by any given class. In order to resolve a class name, the runtime system calls the `loadClass()` method in ClassLoader of the current class. For example, if we have the class:

```
class A {
    public B b;
    public int foo()
        { return b.x+1; }
}
```

The runtime system uses A's ClassLoader to look up the definition of B while type checking `A.foo()`. The reader is referred to McManis [McM96], the Java language specification [GJS96] and the Java Virtual Machine specification [LY96] for more details and examples.

So we see that ClassLoaders work like environments. Each ClassLoader defines its own namespace, so we need multiple environments to model JVM. However, ClassLoaders are also regular objects — each is an instance of a class, and that class has a ClassLoader. (The runtime system's built-in ClassLoader is special in this regard; it does not have a representation as a Java object.) This is why we use first-class environments. User-written ClassLoaders can implement our `lookup` and `bind` operators by returning classes defined by other ClassLoaders in response to the runtime system calling their `loadClass()` method. This was first presented by Chris Hawblitzel [Haw97], but it is not clear if he was aware of the type-soundness problem.

All Java ClassLoaders are required (although this is not enforced) to ask the built-in ClassLoader if it would like to return a class definition before searching themselves for the class definition. In particular, the set of all ClassLoaders in the system forms a tree, because each ClassLoader is loaded in turn by some other ClassLoader. These facts have influenced our (baroque) model of environments.

Our type system is drastically simplified from JVM's. We do not support any notion of subtyping (related to subclassing in the JVM). We use type `dynamic` and the `typecase` expression to model Java's safe cast operator. In JVM, what operations one can perform on an object is a function of the static type of the object. The cast operation walks the class hierarchy via pointer chasing; our `typecase` compares (name, environment) pairs pointwise. Since Java (name, ClassLoader) pairs are in one-one correspondence with classes [Dea97], this is equivalent to our model here. Note that every JVM object has a pointer to its class; this effectively means that all values (of non-primitive type) can be treated as `dynamic`. Where we use `error`, Java would raise an exception.

We use `eval` to start execution in a new environment in PCF$_{\tau,\rho}$. In Java / JVM, we can load a class via a ClassLoader and get back a class object. One of the methods defined by class `Class` is `newInstance()`, so that given a reference to a class, one can create an instance of the class. If the loaded class has a superclass that is visible (in the current ClassLoader), we can cast the result of `newInstance()` to that superclass, and invoke a method of the object, passing arbitrary arguments. (At type `Object`, there is the `equals` method, which takes another object, in case there is no visible superclass.) This starts execution in a class loaded by a different ClassLoader.

Each type identifier is bound to a pair of its definition and a unique stamp. These stamps model the role of `Class` objects in Java. We compare stamps for equality to determine if two type bindings are same (even across environments), much as two objects are instances of the same class in Java when they are instances of the same (defined by reference equality) class.

In Java, we can have values that are instances of a class not nameable in the current ClassLoader. These values must be treated as instances of one of their superclasses; because all classes in Java are subclasses of `java.lang.Object`, we can always treat something as an `Object`. Here, we have values of type dynamic that cannot be matched in the current environment because a sharing constraint is not met between the current environment and the environment where the dynamic value was created. These values can be bound to variables, and passed as arguments to functions, but they can not be taken out of their dynamic wrapper.

Our model of types and environments is not a perfect model of Java; we don't have self-application. However, the Java type system constrains self-application (*e.g.,* through binary methods) so this missing feature does not impact us too greatly.

In summary, PCF$_{\tau,\rho}$ types map to Java classes, environments map to ClassLoaders, type `dynamic` and `typecase` map to Java's checked-cast operation, and `eval()` maps to invoking a method on an instance of a class loaded by another ClassLoader.

4.7 Conclusions

We have studied a fundamental flaw in the JVM definition: *Classes have dynamic scope.* In conjunction with static type checking, this results in type-system failure. In an attempt to fix this problem, the JDK prevents a ClassLoader from defining a class more than once. However, this does not solve the problem of free variable capture in the presence of multiple ClassLoaders. We presented a set of environment manipulation restrictions that solve the general problem. While developing the necessary machinery in PCF, we presented a semantics for first-class environments in a (mostly) statically typed system.

This offers a nice justification for our results in the prior chapter, where our model of dynamic linking provably avoids name clashes. It is a well-known result for λ-calculus that dynamic and static scope are distinguishable only in the presence of a name clash. Such a theorem should be simple to prove given formalizations of type-checking and a small-step operational semantics for Java. The proof proceeds by induction on the length of the typing derivation. Constants have the appropriate type. All rules other than variable use follow immediately from the inductive hypothesis.

Chapter 5

Attacking Cryptographic Hash Functions

5.1 Introduction

Mobile code security has come to rely on digital signatures to authorize mobile code to perform potentially dangerous actions [WBDF97]. If digital signatures can be forged, these architectures will not work. Cryptographic hash functions are a crucial primitive in all known practical digital signature schemes, and are also used in many other protocols, partly because they are freely exportable from the United States. We attack the second preimage problem: given that $F(x_1) = y$, finding $x_2 \neq x_1$ such that $F(x_2) = y$.

Rivest's MD4 [Riv92a] and MD5 [Riv92b] are popular choices for cryptographic hashing, due to their relative efficiency and strength. More recently, SHA-1 [NIS95] has also gained acceptance. While various attacks against MD4 and MD5 have recently been discovered [Dob96], there has not been a general method for computing a second preimage in expected time less than that required by exhaustive search over randomly generated messages. No proposal better than brute force has been proposed for SHA-1 second preimages. This chapter presents an improved second-preimage attack against all three hash functions.

We found our new attack by using the Ever [Hu95] automated reasoning tool, which symbolically analyzes binary decision diagrams (BDDs) describing boolean circuits and finite-state systems.

This chapter is organized as follows: Section 5.2 defines our notation, Section 5.3 describes the attacks, Section 5.4 gives background on binary decision diagrams, Section 5.5 describes our specifications of MD4, MD5, and SHA-1 in Ever [Hu95], Section 5.6 proposes defenses against the attacks, Section 5.7 discusses related work, and Section 5.8 concludes.

5.2 Notation

We write $\mathit{MD_4\,Compress}$ and $\mathit{MD_5\,Compress}$ for the compression functions of MD4 and MD5, respectively, and $\mathit{SHACompress}$ for the compression function of SHA-1. All have the same top level structure, which we write as:

$$\mathit{Compress}(X, \vec{iv}) = \vec{iv} + \vec{H}\,(X, \vec{iv}) \tag{5.1}$$

where X is one block (512 bits), and $\vec{iv}$ is the initialization vector of four or five 32-bit words. The addition is 32-bit unsigned vector addition. We shall frequently refer to $\vec{H}$ on some fixed block, X, which we shall write as $\vec{H}_X$. Following Rivest and NIST, we shall treat $\vec{iv}$ as a four or five tuple, respectively, of 32-bit words, (A, B, C, D, E). Let w, x, y, z range over $\{A, B, C, D, E\}$, f be a bitwise boolean function of three 32-bit words, $Y \lll s$ denote the 32-bit value Y circularly rotated to the left s bits, $0 \leq s \leq 31$, X_i be the i^{th} word of X, $0 \leq i \leq 15$, and k be a 32-bit constant. Then each of the 48 steps of $\vec{H}$ for MD4 can be written as:

$$w \leftarrow (w + f(x, y, z) + X_i + k) \lll s \tag{5.2}$$

Similarly, the 64 steps of $\vec{H}$ for MD5 can be written as:

$$w \leftarrow x + ((w + f(x, y, z) + X_i + k) \lll s) \tag{5.3}$$

Instead of multiple passes over the block, SHA-1 algorithmically extends the block to 80 words, and each step looks like:

$$\begin{aligned} t &\leftarrow (A \lll 5) + f(B, C, D) + E + X_i + k \qquad (5.4)\\ E &\leftarrow D \\ D &\leftarrow C \\ C &\leftarrow B \lll 30 \\ B &\leftarrow A \\ A &\leftarrow t \end{aligned}$$

MD4, MD5, and SHA-1 all divide their input into a sequence of 512-bit blocks. The first block is hashed as described in equation 5.1, where the $\vec{iv}$ is defined as part of the algorithm. To hash the $n + 1^{\text{st}}$ block, the value computed as the hash of the n^{th} block is used as the $\vec{iv}$. These values are called *chaining variables*.

In all cases, the result of $\vec{H}$ is the final values of (A, B, C, D) for MD4 and MD5, and the final values of (A, B, C, D, E) for SHA-1.

5.3 Properties and Attacks

In analyzing various attacks on the hash functions, we will assume that

- the hash function generates b-bit outputs and thus has $N = 2^b$ possible outputs;
- we are given M messages, each of length K blocks;
- time is measured in units of *hashing steps*, where a hashing step is equal to the amount of time required to compute the *Compress* function.

When computing the cost of an attack, we assume that the attacker has already hashed the messages he is attacking, so we do not charge the attacker for performing any computation that is done as part of hashing the messages. The attacker's goal is to produce a message, not equal to any of the given messages, that has the same hash code as any of the given messages.

The key element of the attack is the following insight, which we independently and automatically rediscovered [PvO95]:

Property 5.1 *For any given, fixed input block X, and any given b-bit constant $\vec{c}$, there is a unique value $\vec{j}$, such that $\vec{H}_X(\vec{j}) = \vec{c}$; and this value of $\vec{j}$ can be computed in one hashing step from X and $\vec{c}$.*

That is, given some block X, we can find the $\vec{iv}$ for which that block produces a certain delta. In particular, by choosing $\vec{c} = \vec{0}$, we can find the $\vec{iv}$ for which the block is a fixed point. We independently rediscovered this insight found with automated analysis techniques as described in sections 5.4 and 5.5.

Although this result initially surprised us, a quick glance at equations 5.2, 5.3, and 5.4 shows the reason for this: given a final result for $\vec{H}_X$, there is exactly one predecessor state that can produce the result at each step. Thus, computing the $\vec{iv}$ for which a block is a fixed point (or produces any other constant delta) requires exactly the same number of primitive operations as hashing the block. We call this procedure *reverse hashing*, because it amounts to running the $\vec{H}$ function in reverse.

We can now derive several more properties that follow from Property 5.1.

Property 5.2 *For all X, $\vec{H}_X$ is a permutation.*

$\vec{H}_X$ is clearly an automorphism, and Property 5.1 implies that $\vec{H}_X$ has an inverse. Property 5.2 follows from these two facts.

Property 5.3 *For all $\vec{y}$ and $\vec{z}$, there are exactly 2^{512-b} distinct values X such that $\vec{H}_X(\vec{y}) = \vec{z}$.*

This follows from Property 5.2 by a simple counting argument.

5.3.1 Second Preimage Attack

We now present new methods for constructing second preimages, taking advantage of Property 5.1. Recall that Rivest wrote, "It is conjectured that the difficulty of coming up with two messages having the same digest is on the order of 2^{64} operations, and that the difficulty of coming up with *any* message having a given message digest is on the order of 2^{128} operations." [Riv92b, Section 4] [emphasis added] While this conjecture remains true, strictly speaking, the often assumed extension to second preimages is contradicted.

Single-message case

Before analyzing the general case, we first consider the special case where there is a single message ($M = 1$).

Denote the intermediate *iv*s encountered hashing the message as $\vec{V}_1, \vec{V}_2, \ldots, \vec{V}_K$.

The attack works as follows:

First, generate random blocks X_i. Each X_i is a fixed point for some $\vec{y_i}$: $\vec{H}(X_i, \vec{y_i}) = \vec{0}$. Keep repeating this step until we find a $\vec{y_j}$ equal to some $\vec{V}_p$. The expected execution time of this step is N/K, because each attempt succeeds with (independent) probability K/N.

With a known fixed point in hand, we can now "inflate" the message by inserting as many copies as we like of the block X_j into the message. Since X_j is a fixed point of the compression function $\mathit{Compress}_p$, adding copies of X_j does not affect the hash code of the message.

Simply inflating the message in this way does not give a second preimage, because the hash functions use Merkle-Damgård strengthening: they append the message's length to its end before hashing it. To generate a collision we need to make the length come out correctly. There are several ways to do this:

1. The simplest approach is to repeat the fixed point block 2^{55} times, which adds 2^{64} bits to the input. Since the message length in MD4 and MD5 is computed modulo 2^{64}, this effectively adds 0 to the length field, and the proper hash value comes out. Although this is a seemingly huge blowup, MD4 and MD5 are defined for any input length. The expected time of this step is equal to the time required to emit 2^{55} blocks. Note that this approach will not work for SHA-1, as SHA-1 is undefined for inputs longer than 2^{64} bits.

2. If two of the intermediate values generated in hashing the message are equal, that is, if there exist i and j such that $0 \leq i < j < K$ and $\vec{V}_i = \vec{V}_j$, then we can shrink the message by removing the substring between the ith and jth blocks. After doing this, we can then apply the fixed point attack to grow the message back to its original size. This attack works if an appropriate i and j exist, which occurs with probability approximately $K^2/2N$ for $K \ll N$. Shrinking the message reduces its length to $2K/3$ on average, so this attack takes an expected time of $3N/2K$ hashing steps, when it works. This attack is possible against all three hash functions, though it is very unlikely to work against SHA-1 because of the limit of 2^{64} bits on message length.

3. Regardless of the contents of the message, we can look for a way to shrink the message by seeking a hash coincidence. If $j > K/2$, then we look for a block s such that $Compress(\vec{V}_0, s) = \vec{V}_p$ with $p < j$. Expected time for this step is $\frac{4N}{3K}$ hashing steps, because each attempt is equally likely to generate any of N results, and (since j is uniformly distributed in the interval $(K/2, K)$) the expected value of j is $\frac{3K}{4}$, so on average $\frac{3K}{4}$ of the possible results will allow this step to terminate. Alternatively, if $j \leq K/2$, then we look for a block s such that $\vec{H}(\vec{V}_j, s) = \vec{V}_p$ with $p \geq j$. Expected time for this step is also $\frac{4N}{3K}$ hashing steps, by a similar argument. Total expected running time for this attack, including the time spent finding a way to inflate the message, is $\frac{7N}{3K}$ hashing steps. This attack works against all three hash functions.

Multi-message case

We now turn to the case of multiple messages. In this case, we have a single attack. We first sketch the attack intuitively; then we give a more detailed description that

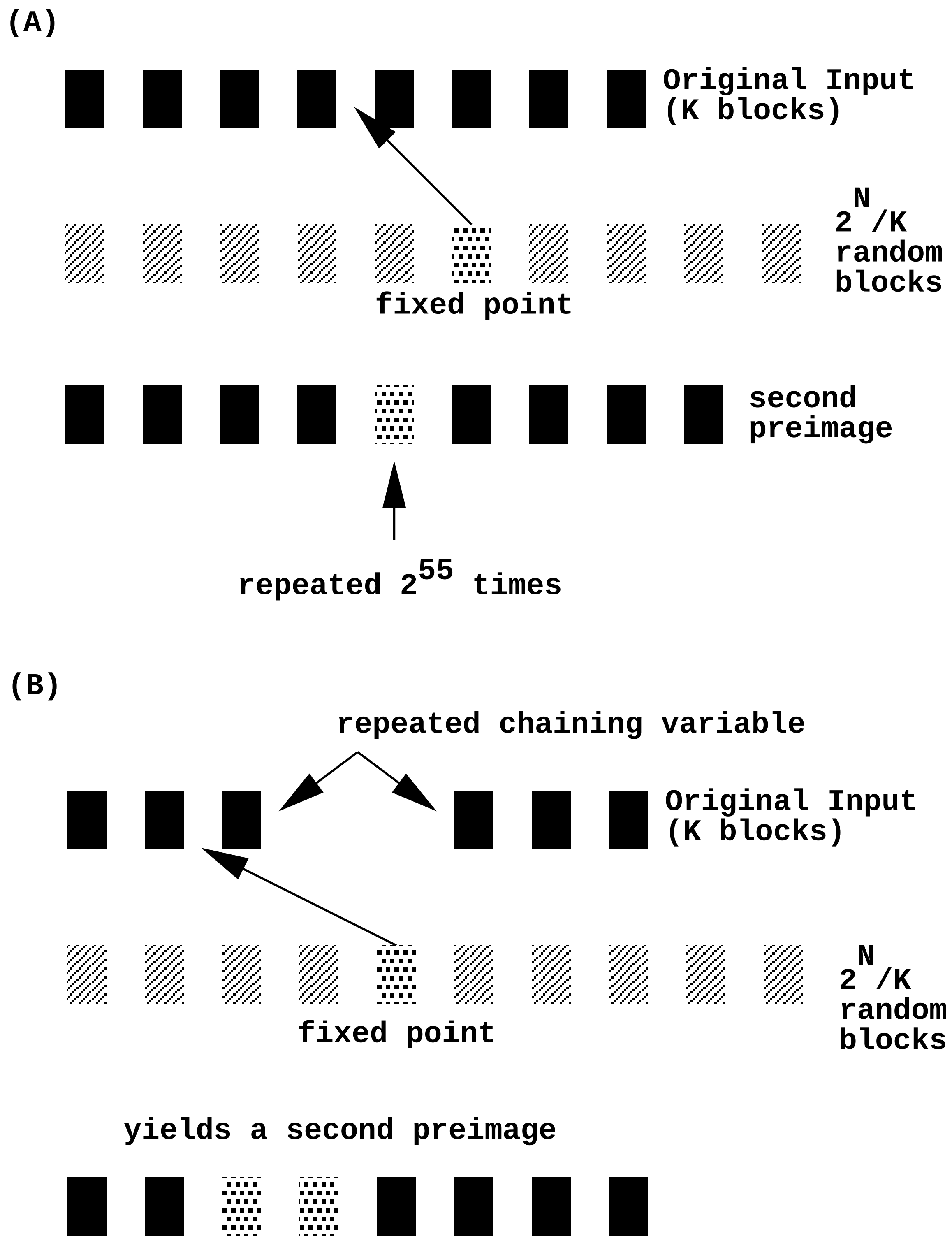

Figure 5.1: Diagrams of the first two second preimage constructions

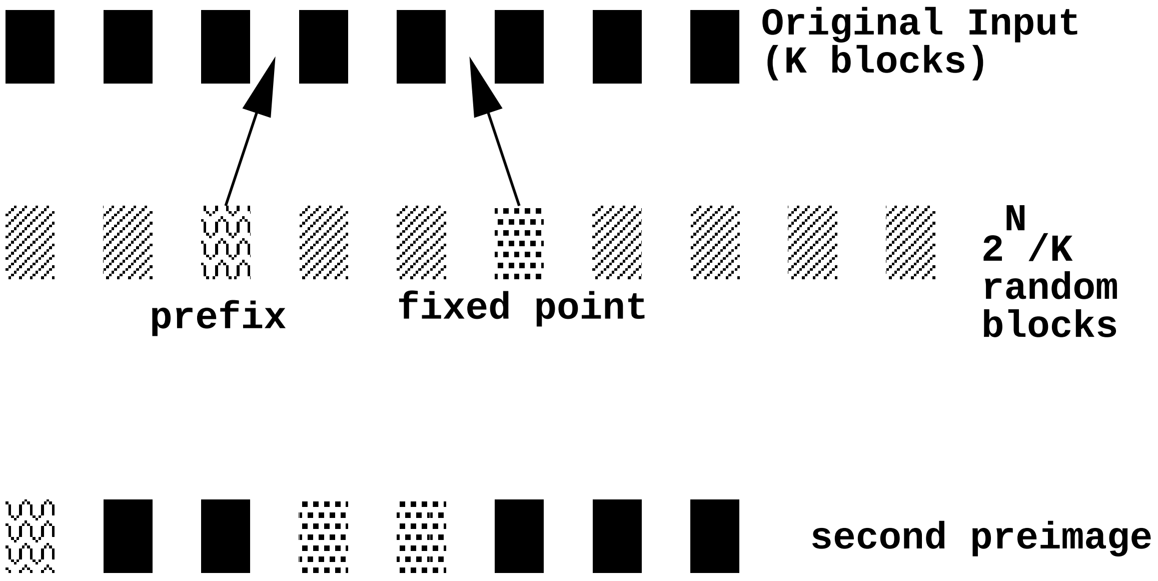

Figure 5.2: Diagram of the third second preimage constructions

includes an analysis of expected running time.

Intuitive Description Intuitively, the attack works by finding a message that can be inflated and then shrunk, to create a message of the same size that has the same hash code. First, we use the fixed point search method to find ways to inflate messages, until we can inflate $\sqrt{2M}$ of the given messages[1]. Then we search for ways to shrink the inflatable messages, by searching for a single block that has the same hash code as some prefix of an inflatable message. (This prefix must stop short of the point at which we can inflate the message.)

Detailed Description The attack goes in two steps:

First, we generate random blocks b_i. For each b_i, we compute the value for which b_i is a fixed point, that is, we find the $\vec{x_i}$ such that $\vec{H}(b_i, \vec{x_i}) = \vec{0}$. If $\vec{x_i}$ matches

[1]We simplify our presentation by ignoring the need for floor and ceiling operators. A correct analysis has slightly different lower-order terms in the running time.

$\vec{V}_p^q$ (the pth intermediate value from hashing message q), call this a *hit on message* q. We continue this step until we have hit on $\sqrt{2M}$ distinct messages. Expected time to generate the first hit is N/KM hashing steps; if we have hit on r unique messages so far, the expected time to hit on one more distinct message is $\frac{N}{K(M-r)}$. The total expected time for this step is thus

$$T_1 = \sum_{i=0}^{i<\sqrt{2M}} \frac{N}{K(M-i)}$$

Mathematical manipulation yields

$$T_1 \leq \frac{\sqrt{2}\,N}{K\sqrt{M}}(1+\sqrt{2/M})$$

Let y_i be the index of the last intermediate value that "hit" in message i, or zero if there was no such hit.

Second, we look for a block s such that, for some a and b, $H(\vec{V}_0, s) = \vec{V}_a^b$ with $a < y_b$. We do this by randomly generating possible values of s until one is found that has the desired property. The expected time of this step is

$$T_2 = \frac{N}{\sum_{i=1}^{i\leq M} y_i}$$

To calculate the expected value of the sum, we note that $\sqrt{2M}$ of the terms will be non-zero, and the expected value of each nonzero term is $K/2$, so the expected value of the sum is $K\sqrt{M/2}$. It follows that

$$T_2 \leq \frac{\sqrt{2}\,N}{K\sqrt{M}}.$$

Adding up the expected cost of the two steps, we get

$$T = T_1 + T_2 \leq \frac{2\sqrt{2}\,N}{K\sqrt{M}}(1+\sqrt{1/2M}) = \frac{2(1+\sqrt{2M})N}{KM} \approx \frac{4N}{K\sqrt{M}} \quad \text{for } M \geq 4$$

Previously, the best known bound for this problem was N/M, achieved by generating random messages and looking for one that hashes to the same value as one of the given messages. Simple algebraic manipulation yields $\frac{4N}{K\sqrt{M}} \leq \frac{N}{M}$ when $K \geq 4\sqrt{M}$. As the intuition suggests, we do better given a relatively small number of relatively long messages. Unfortunately, this attack is still infeasible.

5.4 Binary Decision Diagrams[2]

We found the fixed point attack by using the Ever tool to analyze binary decision diagrams (BDDs). We were attracted to BDDs based on their success in representing complicated functions for formal verification of hardware. Here, we briefly recap some of their key properties, and refer the interested reader to Hu's introductory article [Hu97], and Bryant's survey paper [Bry92] for more background.

BDDs are a compact data structure for representing boolean functions. Conceptually, we start with a decision tree for the function where all variables are mentioned in the same order along each path from the root to the leaves, and each variable appearing at most once on each path. Then, merging equivalent nodes (same label and edges, as in finite state machine minimization), and removing redundant nodes (where both outbound edges lead to the same node), create a minimal directed, acyclic graph. This process is illustrated in Figure 5.3. In practice, BDDs are constructed in reduced form without building the whole decision tree.

Useful facts about BDDs include the existence of efficient algorithms for primitive Boolean operations (*e.g.,* NOT, OR, AND, etc.), that BDDs are compact representations for many functions, such as the parity example in Figure 5.3, BDDs

[2]This presentation closely follows Hu [Hu97].

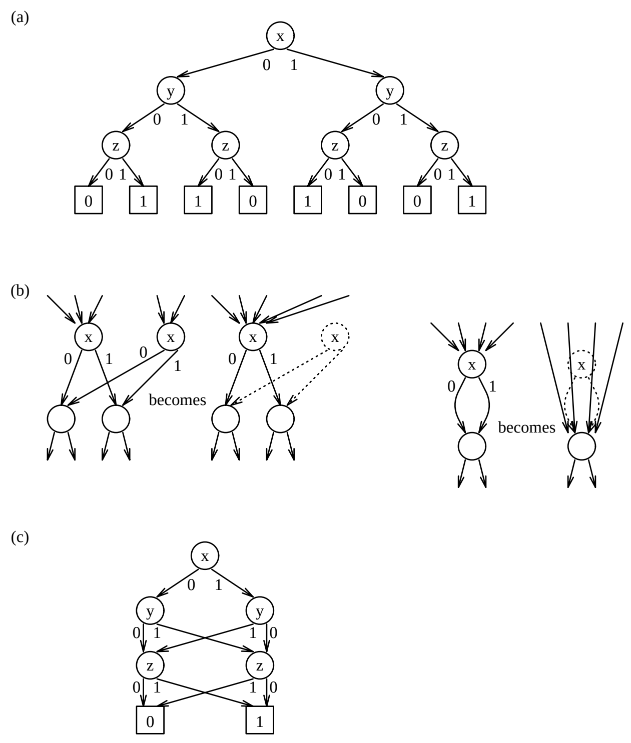

Figure 5.3: Creating the BDD for ($x \oplus y \oplus z$). Courtesy of Alan Hu.

can work in either direction without additional algebra, subject to time and space limitations, and that BDDs are canonical once the variable order is fixed. This means that a tautology can be detected with a simple pointer comparison. BDDs can also be used to represent sets, by using a BDD to represent the characteristic predicate that determines set membership. PVS also represents sets by a membership predicate, although PVS does not use BDDs for this. (PVS does include primitive BDD support.) Set intersection and union become BDD conjunction and disjunction, respectively. Since a function between two sets is just a subset of the Cartesian product of the sets, BDDs can be used to represent functions, as well. This is often used in hardware verification for encoding the transition relation of a state machine.

After manually producing a loop-free implementation of each of the hash functions, the Ever verifier [HDDY92] automatically produced the BDD representation of each hash function. Ever's support for bitwise logical operations, addition, and rotation on 32-bit words was particularly helpful. BDD-size blow-up is an easy trap to fall into, but with care and thought, we were able to use BDDs as an automated, convenient way of manipulating the hash functions.

5.5 Specifying MD4, MD5 and SHA-1 in Ever

The specification of MD4 in Ever follows fairly naturally from the MD4 definition given in RFC 1320. Given the experimental nature of Ever, there are a few minor technical difficulties: Ever does not have functions, so the F, G, and H functions have to be written inline, and the rotations have to be inline expanded to prevent BDD blowup. Writing all the rotations inline requires expanding the code by a factor of four, but this is manageable. The same remarks apply to MD5. The first step of MD4, which Rivest would write as:

$$A = A + F(B,C,D) + X[0] \lll 3$$

is transcribed into Ever as:

```
defpred ABCD_F_3 (x)    -- x is from input
    (compose
        (becomes v.temp^n
                (add
                    v.A^c
                    -- This is F
                    (bor (band v.B^c v.C^c) (band (bnot v.B^c) v.D^c))
                    -- End of F
                    x
                )
        )
        (constrain v.A^n TRUE)
        (compose
            (becomes v.temp^n (<<< v.temp^c 1))
            (becomes v.temp^n (<<< v.temp^c 1))
            (becomes v.temp^n (<<< v.temp^c 1))
        )
        (becomes v.A^n v.temp^c)
        (constrain v.temp^n TRUE)
    );
```

The specification of SHA-1 in Ever was best approached by a different course. SHA-1 repeatedly uses more steps which follow a slightly more regular pattern than MD4 or MD5, i.e., the four additive constants are each used for 20 consecutive steps. The easiest way to generate the Ever specification was to write an AWK script to generate the Ever specification. Other than the minor technical difficulties described above, the only other necessary change was to specialize SHA-1 for a particular input; i.e., we performed constant propagation on the compression function.

Using these specifications, we were able to compute the initialization vector for which the one block message “abc” (appropriately padded and MD strengthened) was a fixed point. The results are shown in Table 5.5. These computations took a few minutes on a 300 MHz Pentium II computer, and required less than 128 MB of RAM. Such a machine is readily available in 1998.

5.6 Defenses

In this section, we examine three possible defenses against fixed point attacks.

Hash	IV

MD4	A = 0xFD48C736
	B = 0x570A4646
	C = 0xB4655DD3
	D = 0x05125A66

MD5	A = 0x627F7E3A
	B = 0x98A8CB26
	C = 0x39423087
	D = 0x5C37CD88

SHA-1	A = 0x4D22C858
	B = 0xF47D6B89
	C = 0x67DC3C45
	D = 0x3B7B9EA2
	E = 0x2DF1C4D2

Table 5.1: Initialization vectors for which the block "abc" (in ASCII, appropriately padded and Merkle-Damgård strengthened) is a fixed point, for each of the three hash functions. These values were computed by Ever.

5.6.1 Strengthening MD Strengthening

The first proposal is extremely simple: MD4 and MD5 both encode the length for Merkle-Damgård strengthening [Dam90] modulo 2^{64}. This means the block length (2^9) divides the modulus, so that we need only 2^{55} blocks of padding. We propose to simply change the modulus to $2^{64} - 1$. Then, a naïve fixed point attack would require 512 times more padding, because the block length would not divide the modulus. For input lengths of $\leq 2^{128}$ bits, the modular reduction is extremely efficient to compute, and in any case, the reduction need only be done once per message. Given that computing the $MD_{4,5}$ hash of a message seems to require serial computation (on a block-by-block basis), there has not been enough time for anyone to hash a message of $\geq 2^{64}$ bits, so all hash outputs computed thus far remain valid. Although better protection is offered by other techniques, this modification is trivial, completely backwards compatible, and has no other effect on the system. Since SHA and SHA-1 do not support inputs longer than the modulus, this change would have no effect on their security.

5.6.2 Combining Function

Dan Boneh suggested looking into the way in which results are chained together. Recall that an MD4, MD5, or SHA-1 initialization vector is an element of $(Z/2^{32}Z)^4$ or $(Z/2^{32}Z)^5$, respectively. We chain the initialization vectors together by simply using the result of the previous block as the $\vec{iv}$ for the next block. Let $\vec{iv}_0$ be the initialization vector specified by the hash function. If we consider hashing a sequence of blocks, $X_0, \ldots, X_n$, where

$$F(\vec{iv}, X) = \mathit{Compress}(X, \vec{iv})$$

using the notation of equation 5.1, this is:

$$F(\ldots(F(\vec{iv}_0, X_0), \ldots,), X_n)$$

Expanding *Compress*, we get:

$$\begin{aligned} \vec{iv}_1 &= \vec{iv}_0 + \vec{H}(X_0, \vec{iv}_0) \\ &\ldots \\ \vec{iv}_n &= \vec{iv}_{n-1} + \vec{H}(X_n, \vec{iv}_{n-1}) \end{aligned}$$

If we then expand each $\vec{iv}_k$ out, the expression for the hash of a message would be one long vector sum. The problem here is that our underlying algebraic structure with addition forms a well-known group. This implies the existence of an identity, and inverses for every element, which make our attacks possible.

We ideally desire that F not have any iterated fixed points, for any values of its parameters. However, for a finite domain, a simple cardinality argument shows that this is impossible. Lacking an ideal solution, we are left with two choices:

1. For F to have fixed points for only a few values of its arguments. A simple example would be:

   $$\mathit{Compress}'(X, \vec{iv}) = q\vec{iv} + \vec{H}(X, \vec{iv}) \tag{5.5}$$

   for a constant $q \neq 0$. Now, $\mathit{Compress}'$ depends on $\vec{iv}$, unless $\vec{iv} = 0$. Property 5.1 appears not to hold, except in the exceedingly rare case that $\vec{iv} = 0$.

2. We can require all iterated fixed points to be long chains that should be hard to cryptanalyze. However, this may conflict with the essential requirement that $\mathit{Compress}'$ behave pseudorandomly, lest it be easily attackable.

Given the choices outlined above, the first may well be more attractive. In any event, we should attempt to avoid combining chaining values with a group operation.

5.6.3 Multiple Hashing

The third proposal, due to Richard Lipton, is also simple. After padding the message and appending the length, we concatenate two copies of the padded message and compute the hash of the concatenation.

This proposal strengthens the hash function by preventing an attacker from making local modifications to the hash function's input. Each block of the message is used at two separate points in the computation; if the attacker tries to modify a block, this will have different effects at the two uses of the block. If the attacker tries to make a controlled change at one point in the computation, this will force another change elsewhere in the computation.

Another view of this method is that we are hashing the message once, but with a variable initialization vector whose value depends on the message itself. Thus any attempt to modify the message will have the effect of modifying the initialization vector as well as the message, complicating the attacker's task.

The drawback of this method is that it doubles the time required to evaluate the hash function.

5.7 Related Work

BDDs have mainly been used for the formal verification of hardware (e.g., [Gup92, Bry95, Hu97] are surveys), although there has been some work model checking software specifications and requirements (e.g., [ABB$^+$96]). To our knowledge, there has been no published work using BDDs for cryptographic applications. Recently, promising work has appeared using automatic, but non-BDD-based, model checkers to analyze cryptographic protocols, while abstracting away details of the underlying cryptographic primitives [Low96, MMS97]. In contrast to that work, we are using automated tools to reason about the primitives themselves – in this case, hash functions.

Since the introduction of MD4 in 1990 [Riv91], there has been much work on its cryptanalysis. Intricate attacks based on the structure of MD4 and MD5 have been proposed by Bosselaers, Den Boer, Dobbertin, and others [Ber92, dBB93, Dob96]. Although some of these attacks use brute force automation to search for solutions to particular subproblems, none consider attempting to automate the analysis of the compression function as we have using BDDs.

5.8 Conclusion

Though the attacks described in this chapter are not practical, in the sense that they do not make it feasible to compute second preimages in a reasonable length of time on today's computers, they do raise some doubts about the strength of popular hash functions. The remedies discussed in section 5.6 appear to mitigate our attacks, but we cannot state with any certainty whether there are related attacks that still work.

We have only begun to explore the power of automated BDD-based analysis tools in determining properties of hash functions. It would be very useful to have some way of characterizing the resistance of various kinds of functions to analysis based on BDDs and similar constructs.

Chapter 6

Conclusions

We examined three applications of formal methods to mobile code security. In the process, we gained new insights into the problem areas. We now understand the problem of Java's dynamic linking to be an instance of the unsoundness of dynamic scope. We have had some success in our exploration of cryptographic hash functions using BDDs, but there remains much work to be done.

The major lesson learned from our analysis of the security of Java implementations is that the problems are most likely to be found in traditionally tricky areas (*e.g.*, exception handling), or areas with innovative designs (*e.g.*, `ClassLoaders`). We are also seeing that "penetrate and patch" is not working because of its classic failure mode: new features introduce new bugs. Unfortunately, this situation appears unlikely to change. Unless market forces shift to reward stable, secure, bug-free software, commercial vendors will continue to emphasize features and short development cycles, which do not have sufficient time to allow the development of even a modest level of assurance.

Java's `ClassLoaders` were a novel and error-prone feature. We defined the proper intuitive notion for `ClassLoader` behavior, the *consistent extension*, and a model for `ClassLoaders` that implemented it. We have fully formal, machine-checked, proofs, for very high assurance. While our model is slightly simplified, it captures the most important cases. Unfortunately, JVM vendors did not exactly follow it, and as a result, we see the security problem we discussed in Netscape 4.05. Hopefully, the lessons we have learned will be more widely applied in the future.

Understanding that the dynamic linking problems were really nothing new, and were just an instance of the general problem of dynamic scope, was a major insight. This justifies the earlier work, because the consistent extension definition implies the lack of name clashes. As usual, in the absence of name clashes, static and dynamic scope are indistinguishable. Why did JVM class loading get defined with dynamic scope? The design was convenient: when class A needs the definition of class B, ask class A's `ClassLoader`. We conjecture is that no one realized what they were implementing dynamic scope at the type level this way. The insight that this is dynamic scope merges the two previously discovered `ClassLoader` soundness issues into the one classic problem of dynamic scope, the capture of free variables. Since capturing free variables can happen in only two ways, we get a nice argument for the completeness of our approach. My work in Chapter 4 can be expanded in many different ways; $PCF_{\tau,\rho}$ is about the simplest Turing complete language[1] that has the features necessary for modeling dynamic linking using first class environments. Finding sufficient safety constraints for a language with a richer type system, imperative features, and concurrency, would be an interesting problem. Java, of course, offers all of these features.

Finally, we shift gears and analyze cryptographic hash functions with the help of binary decision diagrams. Having automatically rediscovered the invertibility

[1] Abadi, *et al.* citeAbadi:1991:DTS show how to encode the Y combinator using type dynamic in simply typed PCF.

of a block acting on an $\vec{iv}$, we reasoned about the behavior of fixed points, and showed that none of MD4, MD5, and SHA-1 offer ideal security against second preimages. BDDs were the appropriate technology to use for this effort; attempting to reason about this in PVS would not have been fruitful. BDDs give us the automation that we need; PVS, like most interactive theorem provers, will only generate a proof if the user already knows why the theorem is true. The user provides a proof sketch, and PVS effectively tracks minor details, and prevents incorrect proof steps. A BDD-based tool, on the other hand, uses heuristics to solve finite problems by brute force. Since each hash function can be implemented for fixed size input by a combinational circuit, the problems are finite and reasonably amenable to BDD-based reasoning. As with all cryptography, cleverness goes a long way. Once we had a method of efficiently computing what $\vec{iv}$ a block was a fixed point for, the actual second preimage constructions were done by hand. Although we cannot effectively compute second preimages, our constructions are better than brute force. This suggests that the security of digitally signed mobile code is slightly less than generally believed, because signatures can be forged faster than brute force. We are continuing to work in this area, exploring new ideas for reducing the BDD size blowup.

All of this work has led to new insight for actually building a secure mobile code system. While language-based protection built on top of a type safe language *should* work, there is a great difference between theory and practice. Since Java did not originally come with a formal specification, many of the difficulties presented by `ClassLoaders` and exception handling slipped through unnoticed. In fact, a long standing known difficulty, dynamic scope, was incorporated into the design (possibly by accident). We have produced a model which rectifies the difficulties with `ClassLoaders`, but it remains to be seen whether the model is simple enough to be robustly implemented in commercially available software.

It is worth noting that Java was developed in an ad hoc style, by non-programming language theorists. Programming language theorists usually write down formal specifications of type systems and semantics while designing a language. They often prove theorems (*e.g.,* type soundness) about the language before implementing it. This sort of work is of more than theoretical interest for language based security: it *is* assurance evidence. The success of Standard ML in the research community contributes to the belief that languages with clean definitions have a bright future, and provides an example to follow for rigorous design. Assurance is a continuum; we can always prove more theorems with more rigor, at greater cost. Java did nothing, with predictable results. We already know how to do much more (as is being done to Java retroactively); it is merely a matter of putting in the work upfront (and getting the support to do so).

Formal methods remain expensive. However, the tools we have used (PVS and Ever) are approachable by the non-expert. An ML programmer should be able to pick up the PVS language very quickly. The Ever specifications of the hash functions are almost straight transcriptions from the C reference implementations. Some knowledge of how to drive these tools is needed to produce useful results. Access to experienced users (or tool developers) greatly speeds the learning process; I have been very lucky in this regard. As always, formal methods focus our attention to the problem at hand, stripping away many inessential details, and thus clarifying the big picture, and the important details.

Formal methods have been effective in the problem domains we have explored. Ever gave us an insight into the structure of three common cryptographic hash functions; knowing that we could compute the $\vec{iv}$ for which a fixed block is a fixed point made us think about how a second preimage could be constructed. PVS forced us to think critically about how dynamic linking works, and ensured the logical correctness of our proofs. Formal methods have provided both assurance that we understand the design of dynamic linking, and insights into the limitations of cryptographic hash functions. In the end, this all leads to a better understanding of mobile code and its security properties.

Ultimately, there is one overarching question beyond the scope of this thesis: Can language based protection be securely *implemented*, in spite of the perils found in large software projects? It is clear that attempting to do this without theory is folly, but even with good theory, the question remains open.

Appendix A

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, strings, etc. Terms include constants, variables, and the forms below. PVS specifications are packaged as *theories*.

Tuples	`(-5, cons(1,null))`
Records	`(# a := 2, b := cons(1,null) #)`
Function Update	`f WITH [(2) := 7]`
Field access	`a(r)`
Tuple deconstruction	`PROJ_`*n*`(t)`
Abstraction	`(LAMBDA(i : nat): i * i)`
Nonempty type	`TYPE+`
Theorems	`THEOREM, LEMMA, CONJECTURE, OBLIGATION` *et al.*

Table A.1: PVS syntax for common functional programming primitives

```
Linking : THEORY
  BEGIN

  IMPORTING string_lemmas, identifiers

  ClassLoader : TYPE+

  primordialClassLoader : ClassLoader

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


ValidClass((c : ClassBase)) : bool =
  CASES c OF
    unresolved(n, r, l) : TRUE,
    resolved(n, r, loader, links) :
        (∀ (cl : ClassBase) :
           (cl ∈ links) ⊃
             loader(cl) = loader(c)∨
               loader(cl) = primordialClassLoader)
    ENDCASES

Class : TYPE = {clb : ClassBase | ValidClass(clb)}

ClassID : TYPE = Ident

ClassList : TYPE = list[Class]

ClassIDMap : TYPE = FUNCTION[ClassID → Class]

ClassDB : TYPE = [ClassID, ClassIDMap]

EnvEntry : TYPE = [string, ClassLoader, list[ClassID]]

ClassTableBase : TYPE = [# env : list[EnvEntry], store : ClassDB#]

ValidCT((ctb : ClassTableBase)) : bool =
  (∀ (e : EnvEntry) :
     (e ∈ env(ctb)) ⊃
       LET y = PROJ_3(e)
         IN
           every(λ (x : ClassID) :
               PROJ_2(e) = loader(PROJ_2(store(ctb))(x))∧
                 x ≤ PROJ_1(store(ctb)),
             y))

ClassTable : TYPE = {ctb : ClassTableBase | ValidCT(ctb)}

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

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

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

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

ct : VAR ClassTable

nm : VAR string

cldr : VAR ClassLoader

cl : VAR Class

every_monotone : LEMMA
    (∀ (p, q : PRED[ClassID]), (y : list[ClassID]) :
       (∀ (x : ClassID) : p(x) ⊃ q(x)) ∧ every(p, y) ⊃ every(q, y))

ll : VAR list[ClassID]

ldr : VAR ClassLoader

FindClassIDswCL((ct : ClassTable), (nm : string), (cldr : ClassLoader)) :
  RECURSIVE
    {ll_ldr : [list[ClassID], ClassLoader]
     |
      (LET (ll, ldr) = ll_ldr
         IN ldr = cldr∧
             every(λ (x : ClassID) :
                 loader(PROJ_2(store(ct))(x)) = cldr,
               ll))}
     = CASES env(ct) OF
      null : (null, cldr),
      cons(hd, tl) :
          LET tab = env(ct), db = store(ct)
            IN IF PROJ_1(hd) = nm ∧ PROJ_2(hd) = cldr
                 THEN (PROJ_3(hd), PROJ_2(hd))
               ELSE
                FindClassIDswCL((#env := tl, store := db#),
                                nm, cldr)
               ENDIF
      ENDCASES
  MEASURE length(env(ct))

FindClassIDs((ct : ClassTable), (nm : string), (cldr : ClassLoader)) :
 list[ClassID] = PROJ_1(FindClassIDswCL(ct, nm, cldr))

every_FindClassIDswCL : LEMMA
  (∀ (cldr : ClassLoader, ct : ClassTable, nm : string, refs : list[string]) :
    every(λ (x : ClassID) :
           cldr =
             loader(PROJ_2(store(ct))
               WITH [(1 + PROJ_1(store(ct))) :=
                 unresolved(nm,
                  refs, cldr)](x))∧
            x ≤ 1 + PROJ_1(store(ct)),
          PROJ_1(FindClassIDswCL(ct, nm, cldr))))

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

define((ct : ClassTable), (nm : string), (refs : list[string]), (cldr : ClassLoader)) :
 [Class, ClassTable] =
 LET cl = mkClass(nm, refs, cldr),
    InsertClass =
     λ ((ct : ClassTable), (nm : string),
       (cldr : ClassLoader), (cl : Class)) :
      LET old = FindClassIDs(ct, nm, cldr),
          newID = GetNextID(PROJ_1(store(ct))),
          newMap = PROJ_2(store(ct)) WITH [newID := cl]
       IN (#env := cons((nm, cldr, cons(newID, old)), env(ct)),
            store := (newID, newMap)#)
  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;

closedWorld : THEOREM
  (∀ ct, nm, cldr :
     LET classloader = loader(PROJ_1(loadClass(ct, nm, cldr)))
      IN classloader = cldr ∨ classloader = primordialClassLoader)

linkClass((ct : ClassTable), (cl : Class)) :
 RECURSIVE [Class, ClassTable]
   = LET getClass = (λ (n : string) : loadClass(ct, n, loader(cl)))
    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)
       ENDCASES
  MEASURE length(references(cl))

linkClass_loader_inv : LEMMA
  (∀ ct, cl : loader(cl) = loader(PROJ_1(linkClass(ct, cl))))

linkClass_name_inv : LEMMA
  (∀ ct, cl : name(cl) = name(PROJ_1(linkClass(ct, cl))))
```

```
resolve((ct : ClassTable), (cl : Class), (cldr : ClassLoader)) : ClassTable =
  LET (newCl, newCt) = linkClass(ct, cl),
      ReplaceClass = λ (ct : ClassTable), (cl, newCl : Class), (cldr : ClassLoader) :
                         LET classDB = PROJ_2(store(ct)),
                             id = PROJ_1(store(ct)),
                             tab = env(ct),
                             clID = FindClassIDs(ct, name(cl), cldr)
                          IN CASES clID OF
                             cons(hd, tl) :
                                (#env := tab,
                                  store :=
                                   (id,
                                    classDB
                                      WITH [hd := newCl])
                                  #),
                             null : ct
                             ENDCASES
   IN ReplaceClass(newCt, cl, newCl, loader(cl));

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)

sysClassTable : ClassTable =
  LET (jlObjectClass, ct1)
       = define(emptyClassTable, "java.lang.Object",
                  null, primordialClassLoader),
      (jlClassClass, ct2)
       = define(ct1, "java.lang.Class",
                  cons( "java.lang.Object", null),
                  primordialClassLoader),
      (jlClassLoaderClass, ct3)
       = define(ct2, "java.lang.ClassLoader",
                  cons( "java.lang.Object",
                        cons( "java.lang.Class",
                              null)),
                  primordialClassLoader)
  IN ct3;

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

proj1_FindClassIDswCL : LEMMA
  (∀ (ct : ClassTable), (nm : string), (cldr : ClassLoader), (classdb : ClassDB) :
    ValidCT((#env := env(ct), store := classdb#)) ⊃
     FindClassIDswCL((#env := env(ct), store := classdb#), nm, cldr) =
      FindClassIDswCL(ct, nm, cldr))

proj1_FindClassIDs : LEMMA
  (∀ (ct : ClassTable), (nm : string), (cldr : ClassLoader), (classdb : ClassDB) :
    ValidCT((#env := env(ct), store := classdb#)) ⊃
     FindClassIDs((#env := env(ct), store := classdb#), nm, cldr) =
      FindClassIDs(ct, nm, cldr))

Resolve : LEMMA
  (∀ (cl : Class), (ct : ClassTable) :
    references(PROJ_1(linkClass(ct, cl))) = null)

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

Monotonic((ct1, ct2 : ClassTable)) : bool =
 (∀ (nm : string), (cldr : ClassLoader), (id : ClassID) :
   (id ∈ FindClassIDs(ct1, nm, cldr)) ⊃
    (id ∈ FindClassIDs(ct2, nm, cldr)))

define_mono : LEMMA
  (∀ ct, nm, cldr : Monotonic(ct, PROJ_2(define(ct, nm, Input, cldr))))

safe_proj : LEMMA
  (∀ ct, (mapping : ClassIDMap) :
    Safe(ct)∧
      ValidCT((#env := env(ct),
        store := (PROJ_1(store(ct)), mapping)
        #)) ⊃
     Safe((#env := env(ct), store := (PROJ_1(store(ct)), mapping)#)))

Initial_Safe : THEOREM Safe(sysClassTable)

loadClass_inv : THEOREM
```

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

loadClass_mono : THEOREM
  (∀ ct, nm, cldr : Monotonic(ct, PROJ_2(loadClass(ct, nm, cldr))))

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

linkClass_mono : THEOREM (∀ ct, cl : Monotonic(ct, PROJ_2(linkClass(ct, cl))))

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

forName_mono : THEOREM
  (∀ ct, nm, cldr : Monotonic(ct, PROJ_2(forName(ct, nm, cldr))))

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

Resolve_mono : THEOREM (∀ ct, cl, cldr : Monotonic(ct, resolve(ct, cl, cldr)))

consistExt : THEOREM
  (∀ ct, nm, cl, cldr :
     cons?(findSysClass(ct, nm)) ⊃
       car(findSysClass(ct, nm)) = PROJ_1(loadClass(ct, nm, cldr)))

END Linking
```

Appendix B

Formal Proofs

We present the PVS proof, first the input script, and then the typeset output, of `loadClass_inv` here to illustrate the formal reasoning taking place.

```
(|loadClass_inv| "" (SKOSIMP*)
 (("" (EXPAND "loadClass")
   (("" (PROP)
     (("" (EXPAND "Safe" +)
       ((""
         (GRIND :IF-MATCH NIL :REWRITES
          ("MapPreservesLength" "proj1_FindClassIDswCL"))
         (("1" (USE "MapPreservesLength")
           (("1" (EXPAND "map" 1) (("1" (GRIND) NIL)))))
          ("2" (USE "proj1_FindClassIDswCL")
           (("2" (EXPAND "ValidCT")
             (("2" (TYPEPRED "ct!1")
               (("2" (EXPAND "ValidCT")
                 (("2" (HIDE -3 1 2)
                   (("2" (INST?)
                     (("2" (GRIND :REWRITES "every_monotone")
                       (("2" (EXPAND "every")
                         (("2"
                           (GRIND :REWRITES "proj1_FindClassIDswCL")
                           (("2"
                             (REWRITE "every_monotone" :SUBST
                              ("q"
                               "(LAMBDA (x: ClassID):
                   loader(PROJ_2(store(ct!1))(car(proj_3(e!1))))
                       =
                     loader(PROJ_2(store(ct!1))
                              WITH [(1 + PROJ_1(store(ct!1))) :=
                                        unresolved(nm!1,
                                                     Input, cldr!1)](x))
                       AND x <= 1 + PROJ_1(store(ct!1)))"
                               "p"
                               "(LAMBDA (x: ClassID):
                  loader(PROJ_2(store(ct!1))(car(proj_3(e!1)))) =
                  loader(PROJ_2(store(ct!1))(x))
                      AND x <= PROJ_1(store(ct!1)))"))
                             (("2" (GRIND) NIL))))))))))))))))))))
          ("3" (USE "proj1_FindClassIDswCL")
           (("3" (EXPAND "ValidCT")
             (("3" (TYPEPRED "ct!1")
               (("3" (EXPAND "ValidCT")
                 (("3" (HIDE -3 1 2)
                   (("3" (INST?)
                     (("3" (GRIND :REWRITES "every_monotone")
                       (("3" (EXPAND "every")
                         (("3"
                           (GRIND :REWRITES "proj1_FindClassIDswCL")
                           (("3"
                             (REWRITE "every_monotone" :SUBST
                              ("q"
                               "(LAMBDA (x: ClassID):
                   loader(PROJ_2(store(ct!1))(car(proj_3(e!1))))
                         =
                     loader(PROJ_2(store(ct!1))
                              WITH [(1 + PROJ_1(store(ct!1))) :=
                                      unresolved(nm!1,
                                                 Input, cldr!1)](x))
                       AND x <= 1 + PROJ_1(store(ct!1)))"
                               "p"
                               "(LAMBDA (x: ClassID):
                  loader(PROJ_2(store(ct!1))(car(proj_3(e!1)))) =
                  loader(PROJ_2(store(ct!1))(x))
                      AND x <= PROJ_1(store(ct!1)))"))
                             (("3"
                               (GRIND)
                               NIL)))))))))))))))))))))))))))))))
```

Terse proof for loadClass_inv.

loadClass_inv:

$$\begin{array}{|ll} & \\ \hline \{1\} & (\forall\, ct, nm, cldr : \ Safe(ct) \supset Safe(PROJ\_2(loadClass(ct, nm, cldr)))) \end{array}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of loadClass,

Applying propositional simplification,
Expanding the definition of Safe,

Trying repeated skolemization, instantiation, and if-lifting,
we get 3 subgoals:

`loadClass_inv.1`:

```
{-1}  nm′ = nm″
{-2}  cldr′ = cldr″
{-3}  null?(map(PROJ_2(store(ct′)),
                PROJ_1(FindClassIDswCL(ct′,
                                       nm″,
                                       primordialClassLoader))))
{-4}  (∀ (nm : string), (cldr : ClassLoader) :
        length(PROJ_1(FindClassIDswCL(ct′, nm, cldr))) ≤ 1)
  |-------
{1}   cons?(map(PROJ_2(store(ct′)), PROJ_1(FindClassIDswCL(ct′, nm″, cldr″))))
{2}   1 + length(PROJ_1(FindClassIDswCL(ct′, nm″, cldr″))) ≤ 1
```

Using lemma MapPreservesLength,

Expanding the definition of map,

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `loadClass_inv.1`.
`loadClass_inv.2`:

```
{-1}  null?(map(PROJ_2(store(ct′)),
                PROJ_1(FindClassIDswCL(ct′,
                                       nm′,
                                       primordialClassLoader))))
{-2}  (∀ (nm : string), (cldr : ClassLoader) :
        length(PROJ_1(FindClassIDswCL(ct′, nm, cldr))) ≤ 1)
  |-------
{1}   cons?(map(PROJ_2(store(ct′)), PROJ_1(FindClassIDswCL(ct′, nm′, cldr′))))
{2}   cldr′ = cldr″
{3}   length(PROJ_1(FindClassIDswCL((#env := env(ct′),
          store :=
            (1
               +
             PROJ_1(store(ct′)),
            PROJ_2(store(ct′))
             WITH
             [(1
                 +
               PROJ_1(store(ct′)))
             :=
               unresolved(nm′,
                 Input, cldr′)])
          #),
          nm″, cldr″))) ≤
        1
```

Using lemma proj1_FindClassIDswCL,

Expanding the definition of ValidCT,

Adding type constraints for ct!1,

Expanding the definition of ValidCT,

Hiding formulas: -3, 1, 2,

Instantiating quantified variables,

Trying repeated skolemization, instantiation, and if-lifting,

Expanding the definition of every,

Trying repeated skolemization, instantiation, and if-lifting,

Rewriting using every_monotone
where q gets $(\lambda\, (x:\ \text{ClassID}):$

$$\begin{aligned}
&\text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}'))(\text{car}(\text{PROJ\_3}(e')))) = \\
&\quad \text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}')) \\
&\qquad \text{WITH } [(1\ +\ \text{PROJ\_1}(\text{store}(\text{ct}')))\ := \\
&\qquad\quad \text{unresolved}(\text{nm}', \\
&\qquad\qquad \text{Input}, \\
&\qquad\qquad \text{cldr}')](x)) \wedge \\
&\ x \leq 1\ +\ \text{PROJ\_1}(\text{store}(\text{ct}'))),
\end{aligned}$$

p gets $(\lambda\, (x:\ \text{ClassID}):$

$$\begin{aligned}
&\text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}'))(\text{car}(\text{PROJ\_3}(e')))) = \\
&\quad \text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}'))(x)) \wedge \\
&\ x \leq \text{PROJ\_1}(\text{store}(\text{ct}')))
\end{aligned}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `loadClass_inv.2`.

```
loadClass_inv.3:

  {-1}  null?(map(PROJ_2(store(ct′)),
                  PROJ_1(FindClassIDswCL(ct′,
                                         nm′,
                                         primordialClassLoader))))
  {-2}  (∀ (nm : string), (cldr : ClassLoader) :
          length(PROJ_1(FindClassIDswCL(ct′, nm, cldr))) ≤ 1)
  |-------
  {1}   cons?(map(PROJ_2(store(ct′)), PROJ_1(FindClassIDswCL(ct′, nm′, cldr′))))
  {2}   nm′ = nm″
  {3}   length(PROJ_1(FindClassIDswCL((#env := env(ct′),
               store :=
                 (1
                     +
                   PROJ_1(store(ct′)),
                  PROJ_2(store(ct′))
                   WITH
                   [(1
                       +
                     PROJ_1(store(ct′)))
                   :=
                     unresolved(nm′,
                       Input, cldr′)])
              #),
              nm″, cldr″))) ≤
          1
```

Using lemma proj1_FindClassIDswCL,

Expanding the definition of ValidCT,

Adding type constraints for ct!1,

Expanding the definition of ValidCT,

Hiding formulas: -3, 1, 2,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of every,

Trying repeated skolemization, instantiation, and if-lifting,

Rewriting using every_monotone
where q gets $(\lambda\ (x:\ \text{ClassID}):$

$$\begin{aligned}
&\text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}'))(\text{car}(\text{PROJ\_3}(e')))) = \\
&\quad \text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}')) \\
&\qquad \text{WITH } [(1\ +\ \text{PROJ\_1}(\text{store}(\text{ct}')))\ := \\
&\qquad\quad \text{unresolved}(\text{nm}', \\
&\qquad\qquad \text{Input}, \\
&\qquad\qquad \text{cldr}')](x))\wedge \\
&\quad x \leq 1\ +\ \text{PROJ\_1}(\text{store}(\text{ct}'))),
\end{aligned}$$

p gets $(\lambda\ (x:\ \text{ClassID}):$

$$\begin{aligned}
&\text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}'))(\text{car}(\text{PROJ\_3}(e')))) = \\
&\quad \text{loader}(\text{PROJ\_2}(\text{store}(\text{ct}'))(x))\wedge \\
&\ x \leq \text{PROJ\_1}(\text{store}(\text{ct}')))
\end{aligned}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `loadClass_inv.3`.

Q.E.D.

Bibliography

[ABB$^+$96] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. In *Symposium on the Foundations of Software Engineering*, 1996.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, 1991.

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, volume 29, pages 13–23. ACM Press, June 1994. Also appears as Princeton University Department of Computer Science Technical Report 452-94, available from `http://ncstrl.cs.princeton.edu/techreports/`.

[And72] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA 01730 USA, October 1972. Volume 2, pages 58–69.

[Bal96] Dirk Balfanz. Personal communication, June 1996.

[Bel95] Steven M. Bellovin. Using the domain name system for system breakins. In *Proceedings of the Fifth Usenix UNIX Security Symposium*, pages 199–208, Salt Lake City, Utah, June 1995. Usenix.

[Ber92] T.A. Berson. Differential cryptanalysis mod 2^{32} with applications to MD5. In R.A. Rueppel, editor, *Advances in Cryptology — Eurocrypt '92*, Berlin, 1992. Springer-Verlag.

[BF96] Dirk Balfanz and Ed Felten. Java security update. *RISKS Forum*, 18(32), August 1996. `ftp://ftp.sri.com/risks/risks-18.32`.

[BL84] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, June 1984.

[BLO94] Guruduth Banavar, Gary Lindstrom, and Douglas Orr. Type-safe composition of object modules. In *International Conference on Computer Systems and Education*, Bangalore, India, 1994. See also: `http://www.cs.utah.edu/projects/flux/papers.html`.

[Bor94] Nathaniel S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *Proceedings of ULPAA*, 1994.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Bry95] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer-Aided Design*, pages 236–243, 1995.

[BSP$^+$95] Brian N. Bershad, Stefan Sava, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the *spin* operating system. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, 1995.

[Car88a] Luca Cardelli. Phase distinctions in type theory. Manuscript available from `http://www.luca.demon.co.uk/Papers.html`, January 1988.

[Car88b] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car96a] Tom Cargill. Personal communication, April 1996.

[Car96b] Tom Cargill. Personal communication, July 1996.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, January 1997. See also: `http://www.luca.demon.co.uk/Papers.html`.

[Cas95] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[CER95] CERT Coordination Center. Syslog vulnerability - a workaround for sendmail. CERT Advisory CA-95:13, October 1995. `ftp://ftp.cert.org/pub/cert_advisories/CA-95%3A13.syslog.vul`.

[Che98] D.M. Chess. Security issues in mobile code systems. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1998.

[Cou95] Antony Courtney. Phantom: An interpreted language for distributed programming. In *Usenix Conference on Object-Oriented Technologies*, June 1995.

[Cro95] Ray Cromwell. Another netscape bug (and possible security hole). Message to the Cypherpunks mailing list (and forwarded to BUGTRAQ); `Message-Id: <199509220612.CAA11441@clark.net>`, September 1995. Available from `http://www.netspace.org/cgi-bin/wa?A2=ind9509d&L=bugtraq&F=&S=&P=499`.

[DA97] Tim Dierks and Christopher Allen. The TLS protocol, November 1997.

[Dam90] Ivan Damgård. A design principle for hash functions. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 416–427. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[dBB93] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of MD5. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 293–304, Lofthus, Norway, May 1993. Springer-Verlag.

[DE97a] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, Paris, January 1997.

[DE97b] Sophia Drossopoulou and Susan Eisenbach. Java is type safe – probably. In *Proceedings of the Elventh European Conference on Object-Oriented Programming*, June 1997.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997. `http://www.cs.princeton.edu/sip/pub/ccs4.html`.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[DFWB97] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browsers and beyond. In *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press, 1997.

[Dob96] Hans Dobbertin. Cryptanalysis of md4. In *Proceedings of the 3rd Workshop on Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–70, Cambridge, UK, 1996. Springer-Verlag.

[FBB$^+$97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OS kit: A substrate for OS and language research. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, 1997.

[Fis96] Kathleen Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996.

[FK97] Michael Franz and Thomas Kistler. A tree-based alternative to Java byte-codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*, 1997. Also appears as Technical Report 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The ssl protocol version 3.0, March 1996. Available from `http://home.netscape.com/eng/ssl3/ssl-toc.html`.

[FM96] Kathleen Fisher and John C. Mitchell. On the relationship between classes, objects, and data abstraction. In *Proceedings of the 17th International Summer School on Mathematics of Program Construction*, LNCS, Marktoberdorf, Germany, 1996. Springer-Verlag. To appear.

[Gen96] General Magic, Inc., 420 North Mary Ave., Sunnyvale, CA 94086 USA. *The Telescript Language Reference*, June 1996. `http://www.genmagic.com/Telescript/Documentation/TRM/index.html`.

[Gho98] Anup K. Ghosh. *E-Commerce Security: Weak Links, Best Defenses*. John Wiley and Sons, February 1998.

[Gib96] Steve Gibbons. Personal communication, February 1996.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GKCR98] R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[GLDW87] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared libraries in SunOS. In *USENIX Conference Proceedings*, pages 131–145, Phoenix, AZ, 1987.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GM84] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[GM96] James Gosling and Henry McGilton. *The Java Language Environment*. Sun Microsystems Computer Company, 2550 Garcia Avenue, Mountain View, CA 94043 USA, May 1996. `http://java.sun.com/doc/language_environment.html`.

[Gos95] James Gosling. Personal communication, October 1995.

[GS98] L. Gong and R. Schemers. Signing, sealing, and guarding Java objects. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 206–216. Springer-Verlag, 1998.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1:151–238, 1992.

[GW96] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, January 1996.

[Has96] Lee Hasiuk. Personal communication, February 1996.

[Haw97] Chris Hawblitzel. Re: Packages, class loaders and security. Usenet `Message-Id: <32EE8704.3B69@cs.cornell.edu>` in `comp.lang.java.security`, January 1997. Available from `http://www.dejanews.com` or this author.

[HDDY92] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Computer-Aided Verification: Fourth International Workshop*. Springer-Verlag, July 1992. Published in 1993 as Lecture Notes in Computer Science Number 663.

[Hen94] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[Hop96a] David Hopwood. Another Java attack. *RISKS Forum*, 18(18), June 1996. `ftp://ftp.sri.com/risks/risks-18.18`.

[Hop96b] David Hopwood. Java security bug (applets can load native methods). *RISKS Forum*, 17(83), March 1996. `ftp://ftp.sri.com/risks/risks-17.83`.

[Hu91] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[Hu95] Alan John Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, December 1995. STAN-CS-TR-95-1561.

[Hu97] Alan J. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 677–682, 1997.

[Jag94] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

[Jan74] Philippe Arnaud Janson. 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.

[JL76] Anita K. Jones and Barbara H. Liskov. A language extension for controlling access to shared data. *IEEE Transactions on Software Engineering*, SE-2(4):277–285, December 1976.

[KLO98] G. Karjoth, D.B. Lange, and M. Oshima. A security model for aglets. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 188–205. Springer-Verlag, 1998.

[LaD] Mark LaDue. Hostile applets home page. `http://www.prism.gatech.edu/~gt8830a/HostileApplets.html`.

[LaD98] Mark LaDue. Applets can create subclasses of applet-classloader in communicator 4.04 and 4.05. `Usenet Message-ID: <352D5749.E7B2C7E1@mindspring.com>` in `comp.lang.java.security`, April 1998. Available from `http://www.dejanews.com` or this author.

[Lam71] Butler W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in *Operating Systems Review*, 8(1):18–24, Jan. 1974.

[Lan81] Carl E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA98*, volume 35, pages 36–44, Vancouver, BC, October 1998. ACM.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: sharing variable bindings across multiple lexical scopes. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 479–492, January 1993.

[LM93] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[LR93] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June / July 1993. Springer-Verlag.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[McM96] Chuck McManis. The basics of Java class loaders. *JavaWorld*, 1(8), October 1996. `http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html`.

[MF96] Gary E. McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, 1996.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8. Elsevier Science Publishers B.V., 1990.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, 1996.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murphi. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Mue95] Marianne Mueller. Regarding java security. *RISKS Forum*, 17(45), November 1995. `ftp://ftp.sri.com/risks/risks-17.45`.

[Mue96] Marianne Mueller. Personal communication, January 1996.

[Nat85] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*. National Computer Security Center, 1985.

[NBF$^+$80] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. A provably secure operaging system: The system, its applications, and proofs. Technical Report CSL-116, 2nd Ed., SRI International, May 1980.

[Neu95] P.G. Neumann. *Computer-Related Risks*. Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-55805-X.

[NIS95] NIST. Secure hash standard. Federal Information Processing Standards Publication, April 1995. FIPS PUB 180-1, US Dept. of Commerce, National Institute of Standards and Technology.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without runtime checking. In *Second Symposium on Operating Systems Design and Implementation (OSDI '96) Proceedings*, pages 229–243, Seattle, WA, October 1996.

[NL98] G.C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.

[OBLM93] Douglas B. Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. Fast and flexible shared libraries. In *Proceedings of the 1993 Summer USENIX Conference*, Cincinnati, OH, 1993.

[Org72] E.I. Organick. *The Multics System: An Examination of its Structure*. MIT Press, Cambridge, Massachusetts, 1972.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes: Language, System, and Prover Reference Manuals.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[PvO95] Bart Preneel and Paul C. van Oorschot. MDx-MAC and building fast MACs from hash functions. In Don Coppersmith, editor, *Proc. CRYPTO 95*, pages 1–14. Springer, 1995. Lecture Notes in Computer Science No. 963.

[R$^+$80] D. Redell et al. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2), February 1980.

[Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical Report A.I. Memo No. 1564, Massachusetts Institute of Technology, Artificial Intelligence Labortory, March 1996.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[Riv91] Ronald L. Rivest. The MD4 message digest algorithm. In A.J. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, pages 303–311. Springer, 1991. Lecture Notes in Computer Science No. 537.

[Riv92a] Ronald L. Rivest. The MD4 message-digest algorithm. Internet Request for Comments, April 1992. RFC 1320; obsoletes RFC 1186.

[Riv92b] Ronald L. Rivest. The MD5 message-digest algorithm. Internet Request for Comments, April 1992. RFC 1321.

[Ros96] Jim Roskind. Java and security. In *Netscape Internet Developer Conference*, Netscape Communications Corp., 501 E. Middlefield Road, Mountain View, CA 94043 USA, March 1996. `http://developer.netscape.com/misc/developer/conference/proceedings/j4/index.html`.

[RRV95] Sreeranga Rajan, P. Venkat Rangan, and Harrick M. Vin. A formal basis for structured multimedia collaborations. In *Proceedings of the 2nd IEEE International Conference on Multimedia Computing and Systems*, pages 194–201, Washington, DC, May 1995. IEEE Computer Society.

[RSS96] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 149–160. ACM, January 1998.

[Sco93] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1–2):411–440, December 1993.

[SFPB96] Emin Gün Sirer, Marc E. Fiucynski, Przemyslaw Pardyak, and Brian N. Bershad. Safe dynamic linking in an extensible operating system. In *Workshop on Compiler Support for System Software*, February 1996. See also: `http://www.cs.washington.edu/research/projects/spin/www/papers/index.html`.

[SM95] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[SR98] Allan M. Schiffman and Eric Rescorla. The secure hypertext transfer protocol, June 1998.

[Str91] Bjarne Stroustrup. *The C++ Programming Langauge*. Addison-Wesley, 2nd edition, 1991.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Sym97] Don Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, Cambridge University, June 1997.

[Uni94] Unisys Corporation. *Unisys A18 System Architecture MCP/AS (Extended): Support Reference Manual*, April 1994. `http://www.unisys.com/marketplace/aseries/pdf/70081781.pdf`.

[Vig98] G. Vigna, ed. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[VS98] D. Volpano and G. Smith. Language issues in mobile program security. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 25–43. Springer-Verlag, 1998.

[Wal99] Daniel S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, January 1999.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, Saint Malo, France, October 1997.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988. Reprinted in *Meta-Level Architectures and Reflection*, P. Maes and D. Nardi, *eds.*, North Holland, 1988.

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 2nd edition, 1983.

[Yel95] Frank Yellin. Low level security in Java. In *Fourth International World Wide Web Conference*, Boston, MA, December 1995. World Wide Web Consortium. `http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html`.