Programming Languages for Mobile Code

TOMMY THORN

Inria/Irisa

Sun's announcement of the programming language Java more than anything popularized the notion of mobile code, that is, programs traveling on a heterogeneous network and automatically executing upon arrival at the destination. We describe several classes of mobile code and extract their common characteristics, where security proves to be one of the major concerns. With these characteristics as reference points, we examine six representative languages proposed for mobile code. The conclusion of this study leads to our recommendations for future work, illustrated by examples of ongoing research.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introduction and Survey; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms: Languages, Security

Additional Key Words and Phrases: Distribution, formal methods, Java, Limbo, mobile code, network programming, Objective Caml, Obliq, object orientation, portability, Safe-Tcl, safety, security, Telescript

1. WHY MOBILE CODE?

The expression "mobile code" has various different meanings in the literature. Just to take three examples, let us cite the following.

—The term *mobile code* describes any program that can be shipped unchanged to a heterogeneous collection of processors and executed with identical semantics on each processor [Adl-Tabatabai et al. 1996].

—*Mobile code* is an approach where programs are considered as documents, and should therefore be accessible, transmitted, and displayed (i.e., evaluated) as any other document [Rouaix 1996].

—*Mobile agents* are code-containing objects that may be transmitted between communicating participants in a distributed system [Knabe 1996].

In this survey we refer to mobile code as software that travels on a heterogeneous network, crossing protection domains, and automatically executed upon arrival at the destination. Protection domains can be as big as a corporate network and as small as a personal hand-held digital assistant. We believe that this characterization is general enough to encompass most usages and

This work was performed under the auspices of Dyade (an R&D joint venture between Bull and Inria).

Author's address: T. Thorn, Inria/Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France; email: ⟨thorn@irisa.fr⟩.

precise enough to exhibit the distinctive features of this technique. For example, it excludes the cases where code is loaded from a shared disk or downloaded (manually) from the Web.

Mobile code supports a flexible form of distributed systems where the desired nonlocal computations need not be known in advance at the execution site. The advantages of this model are many, including:

—*Efficiency.* When repeated interactions with a remote site are needed, it can be more effective to send the computation to the remote site and to interact locally. This is especially the case when the latency of the network is high and the interactions consist of many small messages.

—*Simplicity and flexibility.* The maintenance of a network can be much simpler when the applications are located on a server and clients themselves install them on their sites on demand. Installing new or updated software becomes independent of the nature and number of clients. In some cases, it is even impossible to know in advance all the pieces of code that will be needed at a given site.

—*Storage.* Loading code on demand rather than having all programs duplicated on all sites can significantly reduce the total storage requirement.

We start with a short review of some of the most well-known examples of applications using mobile code.

—PostScript™ is a page-description language designed by Adobe Systems. PostScript is remarkable in that it is also a stack-based programming language and is associated with a large standard library suitable for page rendering. Printing on a PostScript printer consists of composing a program describing the pages to be printed and sending this program to the printer. The printer then executes this program and prints pages as a side effect. This example is a good illustration of some of the benefits that motivate mobile code: the algorithmic description of a complex image can be made very compact and general, independent of printer specifics such as resolution and number of available colors. Also, PostScript printers off-load some of the work burden from the printing host. This compactness, expressiveness, and device independence have made PostScript a de facto standard today.

—Database technology is another area where a form of mobile code has long been used advantageously. The size of a typical database makes it infeasible to transmit it in entirety to a client; thus any database operation must be communicated to and performed by the database server. Today, most commercial databases support the ANSI standard query language SQL for database access. SQL offers a compact notation for expressing complex operations on multiple database relations.

—Documents with embedded executable contents transmitted on the network are another kind of mobile code. Multimedia documents in the Andrew System [Hansen 1990] can include executable scripts written in an object-oriented script language. These enriched documents can be sent as electronic mail or posted as news articles. The scripts are executed under user control and can present interactive dialogues and use graphical facilities. A similar extension for the Internet multimedia standard MIME has been proposed. This extension enables the use of embedded executable content written in Safe-Tcl [Borenstein 1994], a restricted variant of the script language Tcl.

The usefulness of mobile code has also been realized for the World Wide Web. One of the problems with the Web is that interactive pages are impractical in general due to network latencies. Even when latencies are

™ PostScript is a registered trademark of Adobe Systems Incorporated.

not a problem, the content and appearance of Web pages are constrained by what is expressible with the HTML language. Allowing embedded executable contents removes the constraints of HTML and network latency. Among the many existing proposals, the most well known is probably Java by Sun Microsystems.

—Finally, a fourth class of applications of mobile code addresses the problem of software distribution and installation, traditionally known to be costly and difficult. Lucent Technologies has developed Inferno [Lucent 1996], a mobile-code-enabled network operating system aimed mostly at media providers and telecommunication companies. Customer's decoder-boxes or portable telephones running Inferno can be extended dynamically with software in response to their requests. In the same spirit, but in a different context, Sun has proposed the use of Java-capable network computers to replace workstations in corporate networks. The benefit here is the low cost of maintaining an *Intranet* (a local network using Internet technologies) of network computers that download all applications on demand from an application server.

It should be noted that there are subtle differences in the usage of mobile code in the preceding four applications; in the PostScript and the database models, it is the sender of mobile code that takes the initiative of the communication, whereas in the Java and Lucent models, the execution site takes the initiative to load the mobile code. We come back to this distinction when we present programming languages for mobile code.

In this survey we first identify the special concerns for mobile code and their impact on programming languages. In Section 3 we focus on the two most important issues: safety and security. With this background, we examine, in Section 4, six representative languages that have been proposed for mobile code. Section 5 draws the lessons of this study and compares the languages studied. We conclude the article by giving our perspective on the current state of mobile code programming languages and point out directions for future research.

2. PROGRAMMING LANGUAGE CONCERNS

From the application classes of mobile code listed in the introduction, we can extract a number of common needs in terms of programming languages.

—The need for portability. Inherent in the idea of mobile code is the notion of heterogeneous execution sites. It is not possible to have a specific version of the code for every possible architecture, thus the need for portability. The portability issue also has an impact on the kind of services the application can expect from the executing host, for example, trying to write to a file might not be meaningful on a hand-held telephone.

—The need for safety. We use the word "safety" here in the sense that a bug in a safe application should not affect the execution of other independent parts of the environment. Limited assumptions can be made on code imported from an unknown source, which means that it cannot be trusted a priori to be safe, and special protection must be provided. Notice that this definition addresses the safety from an operating system point of view. Safety can also be imposed by appropriate restrictions at the programming-language level. For example, free access to memory can be eliminated through a disciplined pointer model and systematic checking of the bounds for array accesses.

—The need for security. As the mobile code crosses protection domains, special care must be taken to protect against security threats presented by mobile applications. The boundary between safety and security concerns is not always clearcut. Safety is mostly concerned with the behavior of systems in the presence of bugs, but as a lack of safety can be exploited for security breaches, safety becomes a necessary (but not sufficient) prerequisite for security.

We distinguish between four security properties [Russell and Gangemi 1991].

Confidentiality, also known as secrecy: it concerns the absence of leakage of private information (which often occurs through a covert channel, i.e., a channel that is not explicitly intended for communication).

Integrity, also known as accuracy: private data should not be modifiable by unauthorized parties.

Availability, the negation of which is denial of service: the attacker denies normal use of shared resources, for example, by overloading them.

Authenticity guarantees that the identity of a communication partner can be trusted.

—The need for efficiency. Efficiency is almost always an issue for programming languages and their implementations. The special need here is for a minimal overhead for the measures taken to ensure portability, safety, and security.

Portability and efficiency are issues that have been studied in the programming language community for quite a long while. The safety and security issues are well known in the context of operating systems, but safety and, especially, security are issues that have not received enough attention so far in the area of programming languages. Security and safety problems take a new dimension in the context of mobile code. For this reason, we focus on these issues in the next section.

3. SAFETY AND SECURITY ISSUES

Safety and security concern many aspects of a system. We distinguish four levels at which to address these issues: the communication level, the operating-system level, the abstract-machine level, and the programming-language level.

3.1 The Communication Level

In a top-level view of a computer system, we consider a collection of computers connected with some networking technology. Safety concerns here require a robust protocol implementation that can withstand a faulty or malicious communication partner.

For networks like the Internet, where data can pass through untrusted intermediate hosts, the communication itself cannot be assumed to be secret or authentic. Therefore secure protocols based on cryptographic techniques are employed to guarantee confidentiality, integrity, and authentication, even on such networks. This is included in the new Internet protocol, IPv6 [Huitema 1996], but many proposals are also layered on the top of existing protocols. Secure HTTP [Rescorla and Schiffman 1996] and the Secure Socket Layer [Freier et al. 1996] are two examples. Availability is also an issue, but dealing with it is very difficult. This difficulty is illustrated by the many denial of service attacks on the Internet (see, e.g., Fox [1996]).

3.2 The Operating-System Level

Safety and security at the communication level is not sufficient in general. Handling safety and security is also a primary concern at the operating-system level.

Safety is generally ensured through the use of hardware memory protection. This isolates a process from the rest of the system, leaving operating-system calls as the only accessible interface. As no assumptions need to be made on the nature of the process, this leaves a great degree of freedom to the implementation, for example, to choose any programming language available. For mobile code, it has the problem of being very dependent on the operating system and the hardware, and thus not portable. Even when using memory protection is possible, it may not always be desirable.

—Memory protection means that all communications have to cross a protection boundary, for example, using a system call mechanism, and this can be expensive.
—Many smaller systems, for example, personal digital assistants (PDAs), do not have the hardware needed.
—Requiring the use of memory protection makes embedding the mobile code environment in another application much more complicated and often impossible.

Confidentiality and integrity can be achieved by controlling processes' access to information and communication channels. Complete control of covert channels is very hard and rarely attempted in nonclassified systems. A form of availability can be attained by using limits on resources, such as disk space, number of processes, and memory usage, and using preemptive scheduling and timeout in locks. Authentication is usually established through an initial identification of the user (e.g., using a password scheme) and maintained by data structures of the operating system, protected from tampering from user-level processes.

3.3 The Abstract-Machine Level

The safety guarantees obtained through the use of hardware memory protection can also be realized using an abstract machine. Using a language-independent abstract machine retains all the language independence of the operating system solution, but does not have the portability problems. In the simplest version, the protection boundaries are enforced by an interpreter, performing all the needed checks at run-time.

In the Omniware model [Adl-Tabatabai et al. 1996] the overhead of interpretation is eliminated through software fault isolation (SFI). Code for the Omniware abstract machine is translated almost directly into native machine code, but all memory accesses are translated to code that checks for accesses outside a given boundary.

The self-certified code (SCC) [Necula and Lee 1996] technique goes even further and eliminates the overhead of the protection as well. Self-certified code is the pair of machine object code and machine-checkable formal proof. The proof demonstrates that the object code respects the execution site's published (low-level) safety policy. This policy comprises a set of proof-formation rules, along with a set of preconditions. The correctness proof can easily be verified automatically and ensures that the code respects the (low-level) safety and can therefore run without run-time checks.

3.4 The Programming-Language Level

Another way to obtain the required safety is to sacrifice the language independence and use programs written in a safe programming language. Most modern programming languages guarantee against low-level errors through mechanisms such as typing, restricted pointers,[1] automatic memory management, and array bounds checking. It is possible to go even further and use the language scope and access rules to protect the interface of resources. The gain here is that the security implementation, such as resource management and control, can be written in the source language and used as a library.

As an optimization, the high-level program can be compiled and type-checked before being shipped as mobile code. The question then arises as to how to make sure that the object code is really a nontampered output of a correct

[1] Restricted pointers are like references known from, for example, Standard ML. The only valid operations on a pointer variable are dereferencing and assignment.

compiler. Three techniques have been proposed.

—Using cryptographic signatures to reduce the problem to one of trusting the author. As we already trust major software producers enough to run their applications, this can be seen as a continuation of current practice.
—Using cryptographic signatures to trust compilers. The idea is to ship the source to one of a small number of trusted compilation sites for compilation and certification.
—Compiling to an intermediate language that can be (type) checked to verify the same constraints as are imposed on the source language. The success of this approach depends on the intermediate language being suitable for efficient verification and permitting an efficient abstract machine.

These techniques are not exclusive. For example, combining the first two seems easily feasible as they require much of the same technology and infrastructure. Likewise, the abstract machine can include operating-system aspects and can be more or less language-dependent. For instance, depending on the context, we can consider the system libraries either as part of the abstract machine or as part of the language. The languages examined in the following all use combinations of these levels, although this is generally not made explicit.

4. PROGRAMMING LANGUAGES FOR MOBILE CODE

We focus here on a list of representative languages for mobile code. Space considerations prevent us from presenting all the relevant languages. Among the other programming languages for mobile code, let us mention JavaScript [Netscape 1997] and VisualBasic. The interested reader is referred to the bibliography for a more extensive overview of the field.

The first four languages studied, Java, Limbo, Objective Caml, and Obliq, are general-purpose languages, intended for general application development. The last two, Safe-Tcl and Telescript, are special-purpose languages. We expect Java to be the best known language amongst these, and therefore give it a more detailed treatment and use it as the reference point for the other languages.

4.1 Java

Java is a class-based object-oriented language created by Sun Microsystems, with an emphasis on portability and security [Arnold and Gosling 1996; Gosling et al. 1996]. As an example of how to use Java for mobile code, JavaSoft has created the *applet* model. Applets are small applications that are automatically downloaded and executed upon visiting a Web page containing them.

The Language

For clarity, we distinguish three levels in the presentation of Java: the language level, the abstract-machine level, and the library level.

Language level. The language is based on a simplified variant of C++ with all unsafe and most complicated language features removed. The features that have been removed include unsafe operations such as pointer arithmetic, unrestricted casts, unions, and features leading to unmaintainable programs such as the C preprocessor, unstructured gotos, operator overloading, and multiple inheritance. Automatic memory management has been added, guaranteeing against pointer errors due to manual memory management and making usage of dynamic memory much simpler. Array and string types are built in with range check of all accesses. Exception handling has also been added, favoring the creation of robust programs. Finally, for concurrency, Java provides threads and serialized methods, using a mutex-locking on the corresponding object.

Java includes a novel notion of interface types. Interfaces define a collection of abstract methods and constants with their associated types. A class can be declared to implement an interface, in which case it must implement all the abstract methods of the interface. Anywhere a value of an interface type is expected, a value of a class implementing this interface can be used. Interfaces are useful for a number of purposes: they can be used to hide the implementation of a class and to group classes with common functionality without forcing them into a class hierarchy.

Java also uses a notion of *package*. A package groups a number of class and interface definitions. Unlike most module systems, Java packages are open-ended and can be extended with definitions not envisioned by their original creator.

The default visibility of class and attribute definitions can be changed with a visibility modifier keyword. A class can be declared *final*, preventing derivation of subclasses of itself, *abstract*, preventing creation of instances, and *private*, limiting the scope of the class declaration to the containing package. Attributes have four (ordered) levels of visibility: *private, default, protected*, and *public*. Private attributes are only visible from within the object itself, that is, not in objects of a subclass or other objects of the same class. The default visibility extends visibility of the attribute to the package in which it is defined. Protected attributes further extend the visibility to subclasses of the defining class, potentially defined in another package. Finally, public attributes are visible everywhere.

Abstract machine level. The Java Virtual Machine (JVM) is a language-dependent abstract machine that is close enough to Java that its object code can be checked to respect the language semantics. In addition to these static (load-time) verifications, the JVM must implement dynamic checks to guarantee the safety of the language. These are bounds checking on array and string accesses, checking casts to a more specific type, invoking methods on null pointers, and the like.

Library level. Complementing the language, a library provides general-purpose data structures, support for graphical user interfaces, and access to network communication. Applications written using these libraries run unchanged on a wide range of platforms, including Unix, Windows, and Macintosh.

Security

The Java language, as described, is a modern "safe" language, guaranteeing that type and access rules are always respected. This in turn enables a low-level security policy to be expressed within the language itself. The visibility rules for classes and attributes play a crucial role in this respect. Indeed, the interface to local resources is provided by libraries, protected by the scope and visibility rules. Most resources requiring dynamic access control, such as the file system or access to the network, are controlled by a centralized security monitor, called the SecurityManager. The SecurityManager has an abstract type that cannot be instantiated by an applet. All security-related methods are declared *final*, so that applications and applets are forced to use the appropriate code. Without this protection, malicious applets could redefine the method in a subclass, potentially circumventing the security invariants. A final class enjoys even stronger protection, in that the inability to create subclasses also implies the inability to define new methods with access to protected attributes.

Linking

The loading of classes over the network is done by an object of the class *ClassLoader*. This object is created during startup and cannot be replaced by applets afterwards (it is part of the SecurityManager state). As attributes with a default or protected visibility are fully accessible within the package of their definition, these two visibilities would be of little use if applets could introduce classes freely in any package and thus avoid the intended protection. To prevent this, the ClassLoader protects a fixed set of packages from being extended by applets. The exact set is not specified, but includes java and sun. The ClassLoader also maintains a unique name space for each network source, separate from the name space for classes coming from the local file system. Network sources are currently distinguished based only on their symbolic address.

Classes can be loaded from the local file system if they are present in a directory specified in the CLASSPATH variable. This variable is part of the Java environment configuration and can be changed by the user before launching the network browser or Java client, but cannot be accessed or modified by an applet.

As class files loaded through the network cannot be trusted to be untampered with and the abstract machine runs with few type checks, the bytecode is passed through a bytecode verifier that checks that the object code respects the Java semantics: it ensures that the bytecode is in a valid format, that pointers are not forged, that access rules are enforced, that the operand stack is used consistently with respect to the types, and that the parameters passed have the expected types.

Applications

There is no single well-identified application area for the language: any distributed and portable application can take advantage of Java. Sun and Oracle emphasize Java's capabilities as a general-purpose language for corporate Intranets of disk-less network computers. Most commonly encountered applets are animations, games, demonstrations, and interactive multimedia educational programs; but major software houses have already demonstrated complete office application suites written as applets.

Moving beyond applets, let us mention some of the more substantial offerings (some are currently under development) [JavaSoft 1996]:

—*JavaOS*, a complete operating system written in Java, offering portability and extensibility;

—*Jeeves*, a framework for extendible network servers;

—*Java Management API*, a framework for management of heterogeneous networks;

—*Java Electronic Commerce Framework*, a software point-of-sale terminal accessible by any Java-enabled browser;

—*Java Beans*, a component architecture for reusable software components;

—*Java Database Access API*, a uniform interface to relational databases; and

—*Java RMI*, an API for implementing remote method invocation. This will ease the creation of client-server applications and permit the creation of more traditional distributed systems.

Reflections

Java is a promising language with a tremendous market acceptance. Much of this popularity stems from its unique combination of characteristics: close to C^{++}, safe, portable, and concurrent, as well as supplying a rich base library.

Since the first presention of Java, a number of "safety" bugs have been discovered [Dean et al. 1996]. It is of concern that many of the sources of the bugs can be attributed to the vague nature of the definition of Java. Although the core language seems simple, many details are in fact quite subtle.

For example, in Java the integrity of the security depends upon applets not being able to instantiate subclasses of critical classes, like ClassLoader. This condition is checked at run-time by the constructor (a special method devoted to initializing the object upon construction), which throws an exception in case of violation. If the applet can catch this exception *within* the constructor, it has succeeded the instantiation, although the object will only be partially initialized. The subtle restriction imposed on the constructor to avoid these situations was checked by the compiler, but not enforced by the bytecode verifier in an early version of Java [Dean et al. 1996].

Java's current security implementation can only be seen as a first step, as it has a number of shortcomings. For example, as noted in Billon [1996], it currently does not scale beyond simple applets. Many of the prospective applications for Java, such as the ones mentioned in the following, require additional local libraries. Unfortunately, there is currently no way to protect user-defined libraries from redefinitions and extensions from applets. Only the system-defined fixed set of packages is protected. The fact that packages in Java are always extensible makes it impossible to guarantee the security of a package based on its source alone; the semantics of the ClassLoader must be taken into account as well. This seems against the spirit of Java's language-based security, and can be a serious problem considering that the ClassLoaders of the major Java applet environments available today do not have identical semantics [Billon 1996].

We strongly believe that a formal approach to security in Java could help avoid most of these weaknesses and would result in a much cleaner and coherent design. Work on formalizing Java is underway, and progress has been made recently on formal studies of Java's type system [Drossopoulou and Eisenbach 1996].

4.2 Limbo

The technologies of three major industries, entertainment, computing, and telecommunication, are converging. Inferno [Lucent Technologies 1996] by Lucent Technologies (Bell Labs Innovations) is a network operating system designed to suit the constraints and needs of this environment. Inferno is intended to be flexible enough to be employed on devices as diverse as intelligent telephones, hand-held computers, personal digital assistants, television set-top boxes,[2] home video game consoles, and inexpensive network computers. It can also be used on servers such as Internet servers, financial servers, and video-on-demand servers.

The design of Inferno is based largely on Plan 9, a network operating system, also from Lucent Technologies, which emphasizes portability, versatility, and an "economical" implementation. Economical here refers to the computing resources required; Inferno can run within little memory and does not require virtual memory hardware. Portability has two dimensions in Inferno. The operating system itself can run on the bare hardware, or on top of an existing operating system such as Unix, Windows-NT, or Plan 9. In the latter case, the services provided by Inferno are interfaced to the native services of the underlying operating system. The second dimension is the portability of Inferno applications. Applications are written in *Limbo*, an integral part of Inferno, and are compiled to a binary format that is portable between all Inferno implementations. Inferno provides a unified file system interface to operating system services, which hides the fact that the service can be local or remote.

The Language

As for Java, it is useful to distinguish between three levels of Limbo: the language level, the abstract machine level, and the library level.

[2] A set-top box is the consumer receiver and decoder for the (usually scrambled) television signals distributed typically by satellite or cable.

Language level. Limbo is a "safe" imperative language. Its main inspiration is C, but it includes in addition declarations as in Pascal, abstract data types, first-class modules, first-class channels, automatic memory management, and preemptive scheduled threads. It excludes pointer arithmetic and casts.

Abstract data types (ADT) declare objects with variable, constant, and function fields. There is no notion of inheritance for ADTs and no visibility declaration for ADT members. Recursive structures are subject to a few simple syntactic constraints to guarantee that cyclic data cannot be created (recursive fields cannot be assigned). Fields declared cyclic do not suffer from this constraint. The reason for this particularity is the way Limbo's garbage collection handles cycles (see the following).

The declaration of a module identifies the types of exported functions and contains the exported declarations of ADTs, simple type declarations, and constants. In order to use a module, it must be instantiated by loading an implementation of the module (at run-time). The loading is done with the built-in function load that takes a module type and a path to the module implementation and returns the instantiated module (or null if unsuccessful). This allows the program to choose among several implementations of a given module at runtime.

The channels of Limbo allow the communication of any value of its declared type. A channel can be connected directly to another process or, through a library call, to a named destination. Channels are the only built-in primitives for interprocess communication, but more complicated mechanisms can be built upon them.

Limbo's garbage collection is based on reference counting and reclaims the memory of noncyclic data immediately, once the last reference to them disappears. Reference counting has the limitation that it cannot reclaim cyclic data, so cyclic data are treated in a specific way and are eventually reclaimed.

Abstract machine level. Limbo programs are compiled to a RISC-like abstract machine called *Dis*, designed for just-in-time compilation to efficient native code by the Limbo run-time system. Dis does not impose language constraints; for example, Dis code need not be type checkable. Lucent claims that Dis is well suited for real microprocessors and reports excellent performance of their implementation.

Library level. Limbo provides a rich library of standard modules, including modules for network communication, secure and encrypted communication, and graphics.

To reflect the different uses of Inferno, two user interface libraries are available. One, based on Tk [Ousterhout 1994], is intended for "traditional" window-based user interfaces. The other provides ready-made interface components for typical embedded applications, such as interactive TV. The specialized design allows for a minimal memory requirement.

Security

Safety in Inferno is achieved through a safe language with restricted pointers and automatic memory management. Pointers can point to any object but cannot point to inside arrays, and there is no pointer arithmetic or arbitrary pointer type conversion. This safety is not enforced by the abstract machine, though. Instead, Inferno relies on applications being signed by trusted authorities who guarantee their validity and behavior.

Security management in Inferno is inspired by the Plan 9 operating systems; all resources are accessed as files, including data, network communication channels, and the executable modules that constitute the applications. All resources available to an application appear exclusively in the name space of that application. Applications cannot arbitrarily manipulate this name space themselves, but must, for security sensitive resources, call the modules that provide them.

Linking

The built-in support for dynamic linking of modules provides type-safe linking at the user level. Another (type-safe) way to provide a module is to transmit it on a channel of the appropriate type. Currently, certain data types, such as modules, cannot be exported outside a machine [Pike 1997].

Applications

The application domain for Inferno is focused towards applications for service providers. In such environments, only a few, usually fixed, sets of authorities need to be trusted, which justifies the use of cryptographic signatures.

Reflections

Lucent's use of an operating system basis provides a clear separation of responsibilities among the language, the abstract operating system, and the runtime environment. This separation reduces the complexity of verifying security consistency and eases the isolation of security breaches. The module system of Limbo enables a clean and type-safe way to implement dynamic loading.

4.3 Objective Caml

Objective Caml (O'Caml) [Leroy 1997] is a functional language in the ML tradition, originating from Caml, a language developed at Inria that is widely used in education. O'Caml has been used as a language for mobile code in the development of the MMM [Rouaix 1996b, 1996a] Web browser, also developed at Inria. MMM adds the possibility of dynamically linking and executing O'Caml applets accessed through the Web. MMM provides a number of hooks for the applets; for example, applets can add elements to the user menu, include new content decoders, or change the handling of link activation. Applets can access parts of the browser internals, such as the cache and browser navigation values.

The Language

O'Caml includes imperative features, such as references and assignment, and a class-based object system, all integrated within a functional core. The main characteristics of O'Caml and its implementation are:

—*Strong, static polymorphic typing.* The static typing property ensures that "well-typed programs cannot go wrong"; that is, they cannot terminate with a type error. All type errors are caught during compilation. Primitives' errors, like division by zero, are not considered type errors, but are handled through the exception mechanism.
Polymorphic typing allows types to be parameterized over a number of type variables. This makes possible a type-safe construction of general functions. For example, a function calculating the length of a list is not dependent on the type of the list elements. With polymorphic typing, it can be defined once (with type α list $\rightarrow$ int, where α is a type variable) and then used for every kind of list.
O'Caml offers automatic type reconstruction, as is usual with languages in the ML family. For documentation and debugging purposes, it is often useful to manually annotate key functions with their type.

—*A powerful module system.* O'Caml offers a rich higher-order module system in which modules have signatures, providing the names and types of exported elements. O'Caml permits higher-order modules through the notion of functors. A functor can be instantiated by applying it to modules with the same signatures as its formal arguments. The unit for separate compilation is the file, which implicitly defines a module of the same name.

—*First-class (higher-order) functions.* Functions can be passed to other functions or returned as results. They can be anonymous (defined using the lambda notation of lambda calculus), and can refer to variables outside their own definition (free variables). Higher-order functions are at the heart of functional languages, and are thus not specific to O'Caml.

O'Caml includes support for concurrency through threads and mutexes (although applets do not support the use of threads) and class-based object orientation through an extension of the typing discipline. An object is an instance of a static class, which can be the specialization of multiple superclasses. In the case of name classes, only the last entry is visible, but the shadowed name can still be accessed. This is a simple solution, but it entails an asymmetry in which inheriting from A and B is different from inheriting from B and A. O'Caml supports a small number of visibility modifiers for classes and object attributes: a class can be declared virtual, preventing instances from being created, and closed, preventing subclasses from being derived. Attributes can be declared private, making them inaccessible outside the methods of the defining class.

Security

MMM applets are only allowed to use safe variants of the standard libraries. A safe library imports everything exported from the unsafe original, but it only re-exports a selected subset. Entities considered to be safe are re-exported directly, but sensitive structures are exported as abstract data types; for dangerous functions, wrappers are exported that check the capabilities of the applet (a wrapper for a function is another function with the same signature that may perform extra computations before and after calling the original). The capabilities of an applet are represented by an abstract data type with one function to access the encapsulated capabilities and another one to ask the user for extended privileges.

As MMM applets are transmitted in object form, the question of how to trust the binary object code arises. Unlike Java, the object code is not verified before execution, but is instead associated with a cryptographic checksum of the interfaces of the imported modules. For protection from unsafe compilers, the MMM designers suggest employing trusted compilation sites to certify that the object code is the result of a correct compilation.

Linking

O'Caml includes library support for dynamic linking object files. As there is no way to access the loaded entities, dynamically loaded modules are responsible themselves for registering the functions they export. The dynamic linking used for applets constrains the use of the primitives that are considered dangerous.

All applets consist of exactly one (potentially big) module, which may contain nested modules. As the applets are self-contained and only allowed to use a fixed set of development modules, they cannot interfere with one another, thus avoiding the complications of separate name spaces for applets.

Applications

Applications for MMM include browser extensions, decoders for new contents types, animations, games, and the like. The advantage of O'Caml is a richer language, with support for several programming paradigms: functional, imperative, and object-oriented.

Reflections

In Java, since all standard library functions can potentially be called by an applet, they must all be secured. With MMM, only functions exported by the safe libraries need to be checked. The major drawback of MMM is the need for trusted compilation sites.

4.4 Obliq

Obliq [Cardelli 1995] of DEC System Research Center is a lexically scoped, dynamically typed, prototype-based language, designed for distributed object-oriented computations. Computations in Obliq are network-transparent; that is, they depend neither on the allocation site nor on the computation site, but the distribution is managed explicitly at the language level.

The Language

To support network transparency, Obliq extends the static scope to the network: free variables of transmitted computations can refer to objects from the origin site. The language has three main characteristics.

—Any value can be transmitted between hosts, including closures and object references. Objects themselves are local to a site and are not considered as values, but object migration can be programmed with a combination of closure transmission, aliasing, and object cloning (see the following).

—Obliq belongs to a class of object-oriented languages called *prototype based*. In prototype-based languages there are no classes, and objects are created by copying (cloning) existing objects (the prototypes). Obliq uses a simple variant of prototyping, called *embedded* prototyping, which avoids all the complications of delegation-based prototyping [Malenfant 1995]. In embedded prototyping, all the methods valid on an object are contained in the object itself; that is, they are not searched for in a list of superclasses.

—Obliq is dynamically typed. Type errors are caught cleanly and propagated to the origin site.

An object in Obliq is a collection of attributes (named values). A simple point object p can be written {x => 3, y => 4}. There are four basic operations on objects.

Selection/invocation: using the value of an attribute or invoking a method, for example, p.x and display.plot(p).

Updating/overriding: changing the value or the method bound to an attribute, for example, p.x <– 4 and display.plot <– lineto. Notice that it is legal to change a value into a method and a method into a value.

Cloning: creating a shallow copy of an object. The immediate values of attributes are copied, but structured values introduce sharing. For example, array elements are shared between the clone and the original object. Cloning is generalized to support mixing several objects with disjoint names. Using the given examples, clone(p,display) produces an object with at least the attributes x, y, and plot.

Aliasing: mechanism of aliases redirecting attributes to attributes in other objects. All selections and updates on an aliased field are done on the redirection target. For redirected method invocation, the "self" object is the object containing the redirected target, not the object containing the alias. An alias itself can redirect to another alias. Objects consisting only of aliases are called surrogates (also known as proxies in other languages). For examples of aliasing and redirection, consider {x => alias x of p1} and redirect p2 to p end. The latter makes all attributes of p2 aliases of the corresponding attribute of p.

Objects can be protected against modification, aliasing, and cloning from outside the object using the protected keyword. Safe interfaces to objects can be constructed through a combination of protection and surrogates.

Concurrency is inherent in Obliq; processes can execute independently on distinct servers and processes can spawn new threads locally. To handle concurrent accesses, Obliq supports serializing objects. An object is serialized if at most one thread can access an object or run one of its methods at any given time. This is realized using a mutex on the object, which is acquired when one of its methods is invoked and released when the method returns. To avoid trivial deadlocks, operations and method calls from within the object itself are not subject to locking. For details, see Cardelli [1995].

Lexical scoping hides named values from outside a given block and run-time typing ensures that these scope rules are enforced. Extending the lexical scope to the network makes it possible to use scope rules to address security issues, such as information hiding; a procedure executing on a foreign server has only access to its own parameters and free variables. Communications between two independent servers are meditated by a shared global name server, which allows servers to import and export local values. To have a procedure executed on a distant server, the name server is asked for the *engine* object accepting procedures. For example, remote invocation can be programmed as follows (on the client side):

```
let mydisp = net_import("display", Namer);
             mydisp.plot(p);
```

Here the name server, Namer, is asked for the value registered with the name display. After this call, mydisp is a reference to the display object, either locally or on a remote server, and is treated like any other object. For example, we can invoke the method plot with a variable p. This example assumes that some process has exported display with

```
net_export("display", display)
```

Object migration can be programmed using a combination of closure transmission, cloning, and surrogates. The following example is taken from Cardelli [1995].

```
let migrate
  Proc = proc(obj, engineName)
    let engine = net_importEngine
      (engineName, Namer);
    let remoteObj = engine(proc(arg)
      clone(obj) end);
    redirect obj to remoteObj end;
    remoteObj;
end;
```

To migrate an object obj to an engine, we first ask a name server for a reference to the engine. Next, we remotely execute on this engine (engine(. . .)) a cloning operation of obj, resulting in a remote object.[3] Finally, all attributes of obj are made aliases for the corresponding attributes of the remote object (redirect obj to remoteObj end).

The distributed object model is closely based on (and implemented with) the Modula-3 network objects [Birrell et al. 1993].

Security

Besides the basic use of scope to control what is exported, no special provision for security in Obliq is made at the time of writing [Cardelli 1995].

Linking

Transmitted closures can use functions from the basic library, but do not otherwise gain access to names from the receiving site. Names are explicitly exported by passing them as parameters to the received closure.

Applications

Obliq is an academic project, and the collection of example applications is small. Examples include a user-interface toolkit, algorithm animation, 3D graphics, and its use as the basis of the Visual Obliq distributed application builder.

[3] Note that the engine-specific information supplied as parameter arg to the migrated closure is not used here.

Reflections

Using network-wide scope for distributed applications leads to an elegant and powerful model of programming. As object migration can be expressed within the language, it is possible to program autonomous traveling agents in Obliq. This is not possible in the model employed by Java and O'Caml. Without further experimental results, however, it is difficult to evaluate the advantages and drawbacks. It seems that this model could be inefficient, leading to many small messages to be transmitted: one for each access to a remote object. Also, Obliq seems to require a much tighter coupling of hosts to support a distributed garbage collection.

4.5 Telescript

Telescript [General Magic 1996] of General Magic is an object-oriented class-based language designed for network programming. Telescript is intended not as a general-purpose language but as a specialized language for communication, just as PostScript is a language for printing. The system is based on a number of metaphors from the real world. The central concept in Telescript is the *agent*, which autonomously travels on the *Telesphere* (a Telescript network of engines) doing business on behalf of its owner. The engine is a Telescript interpreter with a collection of built-in classes and an engine *place*. Engines provide persistence of objects, even in the presence of a system crash. Places are stationary processes that can accept incoming traveling agents. Users can create their own places nested within other existing places. Resource usage can be tracked and charged to the responsible user.

The Language

The Telescript language itself is class-based and includes run-time typing. Classes can inherit from a single superclass and any number of *mix-ins*, abstract classes that cannot be instantiated. Mix-ins can themselves inherit from other mix-ins.

Use of classes can be restricted in two ways: a sealed class cannot be specialized, and an abstract class cannot be instantiated. Attributes of objects can be either private or public. Private attributes can only be accessed from the class itself and its subclasses, whereas public attributes are unrestricted. The operator protect, a novelty of Telescript, turns object references into protected references. A protected reference cannot be used to modify an object.

Agents are *processes* with a number of properties.

—The *telename* is the pair of an *authority* and a process identity that together name a process. The authority identifies the (usually human) Telescript user.
—The *owner* is the process that will own future created objects (except processes, which own themselves). This is usually the current process, but it can be temporarily changed. Objects not owned by any process are garbage collected.
—The *sponsor* is the process whose authority will be attached to and charged for new created objects.
—The *client* is the object whose code requests the current operation.
—The *permit* specifies the capabilities of the current process. A permit has a number of process parameters:
  —the *age* is the maximum life in seconds,
  —the *extent* is the maximum size of memory allowed to the process,
  —the *priority* is used to determine when to schedule the process for execution, and
  —the Boolean parameters *canCreate*, *canGo*, *canGrant*, and *canDeny* specify whether the process can create new processes, travel, raise the permission level of other processes, and lower the permission level of other processes, respectively.

Agents are sent by invoking their go operation with a ticket, specifying the destination place and possibly the route to this address. If the destination accepts the agent's authority and permits, the agent is sent together with its objects to the place and resumes execution within the new place. The effective capabilities of a process are computed as the intersection (minimum) of the four permits process, local, regional, and temporary. The local permit is imposed by the place entered, the regional permit is imposed by the engine, and the temporary permit can be imposed by the language construction restrict. In the following we give an example (taken from General Magic [1996]) of how to execute a method from an untrusted object, using a temporary permit.

```
paranoid := Permit( );
  paranoid.canCreate = false;
  paranoid.canDeny = false;
  paranoid.canGo = false;
  paranoid.canGrant = false;
  paranoid.age = *.age + 2;
  paranoid.extent = *.size + 1000;
  try {
    restrict paranoid {
      yourObject.yourSuspect-
      Call( );
    };
    catch   failed:   PermitViolated
    {. . .};
    catch . . .
  }
```

First we create an new empty permit, named paranoid, which we initialize with very restrictive permissions. We set the maximal age to current plus two seconds and then allow it to allocate 1,000 bytes of storage. The suspicious call, yourObject.yourSuspectCall( ), is then executed in a try block using the paranoid permit. The try block enables us to catch violations, such as code running too long or using too much space.

Four built-in mix-ins are available for further protections on classes:

—*unmoved*: objects of this class cannot be taken along with a traveling agent;
—*uncopied*: objects of this class cannot be copied;
—*copyrighted*: objects of this class can only be instantiated if authorized by a Copyright Enforcer object; and
—*protected*: objects of this class cannot be modified.

Security

As is apparent from the preceding, security is an overall consideration that affects most of the Telescript language. The permission model is elaborate and applies to resource consumption as well (the model can thus address denial of service issues).

Linking

Mobile processes in Telescript are run in a separate domain and can only interact directly with the engine in which they run. All interprocess access is mediated by the engine.

Applications

Telescript is envisioned to make possible an electronic marketplace where users can launch their agents to search and reserve tickets, inspect currencies, and the like.

Reflections

The Telescript system includes a number of features to restrict the actions of agents, but they seem to suffer from a lack of systematic design. It is not clear how to be convinced of the consistency of the implemented security restrictions.

A positive aspect of Telescript is that it tries to deal with denial of service attacks. Telescript agents have their own initiative to travel and are thus more powerful than Java applets, but in a sense, also more dangerous: they can be hard or impossible to control once launched. An interesting aspect of Telescript is that the user does not have to be connected to the network while his agent is acting. The agent can finish its business and return to the user once he reconnects to the network.

4.6 Safe-Tcl

The idea of executable contents had been realized in the context of electronic mail before the arrival of the World Wide Web. We present Safe-Tcl, the most popular among several similar proposals. First Virtual Holdings proposes Safe-Tcl as an extension to MIME, the Internet multimedia mail standard [Borenstein 1994]. The MIME standard defines a standard encoding for enriched mail; that is, mail with more than just ASCII text. MIME mail consists of several parts, each of which can have a different *contents type*. The simplest contents type is just the ASCII text, but contents types include several popular formats for pictures and sound. Safe-Tcl is proposed as an executable contents type of MIME, and thus as the standard language for executable contents within email.

Reflecting the *store and forward* nature of electronic mail, three different execution phases are distinguished: delivery-time, receipt-time, and activation-time. Delivery-time is when the mail leaves the sender, receipt-time is when the mail arrives at the destination, and activation-time is when the user reads the mail. It is specified in the MIME header in which of the three phases the program is intended to be executed. (These three phases coincide for Web pages.)

The Language

Safe-Tcl is, as the name implies, based on Tcl [Ousterhout 1994], a procedural script language designed to be simple, portable, easily embedded, and powerful. Efficiency was a minor design issue. For simplicity, every value in Tcl is represented as a string, including programs themselves. Scope rules are very simple in Tcl; there are only two scope levels: local (to a function) and global.

Security

Tcl is already a safe language in the sense that there is no notion of pointers, casts, or unchecked array accesses. The aim of Safe-Tcl is to be a safe and secure programming language. To achieve this, every language construction and primitive of Tcl was carefully examined. Primitives considered to be too dangerous or general were replaced by a collection of more specific ones. For example, the file system access functions were removed and replaced by an isolated global configuration space. This space is accessed using two functions: SafeTcl_setconfigdata and SafeTcl_getconfigdata. The former associates a string to a key, and the latter returns the string associated with a key. A rich but safe graphical user interface was a major concern in the design of Safe-Tcl. This has likewise been achieved by replacing primitives that are too general with more specific ones.

Linking

As the Safe-Tcl environment is likely to have a great deal of its implementation in Tcl, two interpreters are used: one only for Safe-Tcl, running the untrusted applications, the other for full unrestricted Tcl. The untrusted application can interact directly only with the Safe-Tcl interpreter.

Applications

Typical applications reported for Safe-Tcl include advanced user dialogues for ordering and voting. Safe-Tcl has also been used experimentally for applets in the SurfIt! Web browser [Ball 1996].

Reflections

Safe-Tcl's inherent limitations in terms of efficiency place it in a weak position in the competition. On the other hand, the Safe-Tcl language is much smaller than any of the preceding five, known to be easy to embed, and requires only a small amount of storage.

Table I. Programming Language Features

Language	OO	Concurrency	Mobility	Safety	Security model	Trust in the object code
Java	√	√	Fetch	√	PL	Verified object code
O'Caml	√	Some	Fetch	√	PL	Signed object code
Limbo		√	Fetch	√	OS	Signed object code
Obliq	√	√	Agent	√	PL	No provision
Telescript	√	√	Agent	√	PL	Secure network
Safe-Tcl			Fetch	√	OS	Not applicable

5. REVIEW AND COMPARISON

The main features of the languages presented in Section 4 are summarized in Table I. Let us now review them and use them as the basis for a comparative study.

—*Object orientation.* In the context of mobile code, objects are a convenient entity in which to encapsulate data and programs to be sent on the network. They also serve as entities for grouping information with the same access restrictions.

—*Concurrency.* In a distributed context, the notion of simultaneous and independent computations is a natural one. For Java and O'Caml, support for concurrency is rudimentary; multiple threads of control and corresponding serialization are supported (O'Caml applets do not support concurrency, though). Limbo and Telescript add a rich support for network communication. Limbo includes channels as first-class values. Concurrency is also inherent in Obliq.

—*Mobility.* We can distinguish two different models of mobility.

  —We call *code fetching* (noted by "Fetch" in Table I) the model used by Java, O'Caml, and Limbo in which the user *downloads* the code to be executed. The initiative is with the *receiver* of the code.

  —Mobile agents (Obliq and Telescript) are processes that can be programmed to migrate themselves, so the initiative is with the mobile code itself. We denote this model "Agent" in Table I.

—*Safety.* All of the studied languages are "safe" in the sense that access boundaries expressed in the language are enforced. This is an essential property for a language if security issues are to be expressed using language constructs. Enforcing safety involves ruling out pointer arithmetics, checking array bounds, using automatic memory management, disallowing arbitrary casts, and dynamically checking casts that promote objects to more specialized classes.

—*Security model.* The implementations of access control can be roughly classified in two categories: the operating-system approach and the programming-language-based approach (noted, respectively, as "OS" and "PL" in Table I). In the former, the capabilities of applets are hardwired into the run-time system of the language, and in the latter, they are programmed in the language using protection features such as scope and visibility rules:

  —Java has trusted libraries with access to native code functions. The trusted libraries are protected through a mixture of language constructs and sensitive functions call upon a (protected) security monitor (the SecurityManager) to check dynamic access control.

  —O'Caml uses the module system to restrict applets' use of libraries to a fixed safe subset. This subset provides a supervised interface to the underlying operating system. The security monitor uses a variable of an abstract data type to represent

Table II. Class and Attribute Protection and Keyword Used

Protection offered	Java	O'Caml[a]	Obliq[b]	Telescript	Limbo
		Class protection			
No subclasses	final		protected	sealed	
Subclasses cannot add methods		closed			
No instances	abstract	virtual	protected	abstract	
Visible only in same package	private	NA[c]	NA	NA	
No outside updates			protected		
No aliases	NA	NA	protected	NA	
Mutual exclusion			serialize		
		Attribute protection			
No restriction	public	default[a]	default	public	exported
Visible only in the same package or in subclasses	protected	NA	NA	NA	NA
Visible in subclasses	default		NA	private	NA
Visible only in the defining class	private	private	NA		default
Runtime protected reference				protect	
Mutual exclusion	synchronized				

[a] Visibility in O'Caml can furthermore be restricted using the module system.
[b] Object used as prototypes play the role of classes in Obliq. The restrictions apply only to operations from outside the object.
[c] Not applicable.

its capabilities.

—Limbo provides all available resources as files in an applet-specific file-system hierarchy, separated from other applets. New resources are obtained through system modules. Limbo offers no support for protecting user-written modules.

—Obliq has the language constructs necessary to program access control in the language. Examples on how to do this are given in Cardelli [1995], but there is very little detail on how this is exploited by the Obliq system itself.

—In Telescript, capabilities are represented by protected permission objects.

—Safe-Tcl offers a fixed and restricted functionality through the built-in functions.

—*Trust in the object code.* The safety of object code is based either on trust or (in the case of Java) on verification. In the case of O'Caml and Limbo, the trust is based on a cryptographic signature of a trusted authority. Telescript agents are trusted based on their origin as the network is "secure" and sender addresses can be trusted to be correct.

As the security policy for the object-oriented languages is implemented using objects, it is interesting to compare the possibilities for restricting access to part of the objects in the different languages. Table II summarizes the visibility rules for the four object-oriented languages studied in Section 4. We have included a column for Limbo, whose first-class modules can be compared with classes.

6. PERSPECTIVES

The informal treatment of both language and security aspects is a major drawback of all the languages studied. Mobile code is executed within a complete environment (the run-time environment of the language, the Web browser, the operating system, the network, etc.), so arguing about security enforcement is meaningless without a clear specification of the separation of the responsibilities among the various entities of the environment (what entity is assumed to ensure what property?). A number of the flaws discussed in Dean et al. [1996] can be seen as a consequence of the lack of such a clear separation. For example, in Java, classes loaded from the local file system are more trusted than classes loaded through the network, and thus the former have access to more dangerous operations. Here the integrity of the system depends on both the local operating system and on the Java system. One attack exploited a flaw that made it possible to load classes from anywhere in the file system [Dean et al. 1996]. For this attack to succeed, it must be possible for the attacker to upload a file somewhere on the victim's file system. This can often be done in a variety of ways, depending on the local operating system. Another attack allowed applets to connect to arbitrary hosts. The attack succeeded due to a weakness in the Java library, where an external name service was implicitly assumed to be trustworthy, which in fact it was not.

Current work on mobile code does not take enough account of the research done in programming language semantics [Nielson and Nielson 1992; Schmidt 1986], formal methods in software engineering, like VDM [Bjørner 1991a,b], Z [Spivey 1988], RAISE [Brock and George 1990], and B [Lano 1996], formal models of security [Bell and LaPadula 1973; Landwehr 1981; McLean 1994], or research on static program analysis [Banâtre et al. 1994; Denning and Denning 1977; Mizuno and Schmidt 1992; Volpano et al. 1996]. As a starting point, a semantic definition of the language would provide an important insight and emphasize the weak parts of its definition with respect to security. Such a definition would also make possible formal statements for the security claims made by the proponents of the language. Having a semantics for the language would not be enough, though. Security is a global property, so a security model must take into account all aspects of the system supporting the execution of the code. This includes in particular the hardware, the operating system, the abstract machine, the module libraries, the security manager, and the browser. A security weakness in just one of these endangers the security of the whole system.

Existing research into formal methods for security [Bell and LaPadula 1973; Landwehr 1981; McLean 1994] provides a strong foundation on which such a model should be build. The challenge consists of integrating the different levels mentioned in a coherent and useful way. The formal model will be useful only if it can support the expression of the global security policy and its decomposition to highlight its impact on the various components of the system. One of the components is the execution model of the mobile code language, which would then be characterized precisely in terms of security requirements. This, in turn, would provide the necessary formal basis for both static and dynamic verifications in the language for mobile code.

Several efforts have recently been undertaken that suggest promising avenues for future research to provide a formal basis for mobile code verification. Among them, let us mention:

—Mizuno and Schmidt [1992] derive a security flow analysis as an abstract interpretation of an enriched denotational semantics. The analysis is compositional; individual modules can be analyzed separately and the results (symbolic expressions) can be combined to obtain an analysis of the entire system.

—Banâtre et al. [1994] present the development of a static analysis for information flow in a simple guarded command language. The information-flow logic is based on a noninterference property. The analysis is then derived through successive refinements of the proof system into a complete algorithm for information-flow analysis.

—Volpano et al. [1996] present a sound type system for secure flow analysis. The multiple sensitivity level of Denning's lattice model is formulated as a subtyping relation that can be statically checked. The soundness of their system is formulated as a noninterference property of well-typed programs.

—Necula and Lee [1996] present a technique for safely loading binary extensions into an operating-system kernel. The technique is based on pairing the object code with a machine-checkable proof that the object code respects the published safety policy of the operating system. The proof, formulated in terms of a simple type system, is verified together with the object code by a type checker in the operating system. If it is found to be correct, the code is allowed to execute without any dynamic check whatsoever.
Their approach represents the best that can be achieved in terms of performance (no more overhead is incurred after the type checking) without being dependent on cryptographic signatures. The major problems with their technique is the burden of proof generation, which is manual, and the fact that each safety policy potentially requires its own proof.

—Borgia et al. [1996] present a structural operational semantics for the Facile language based on the notion of proved transition systems. Facile is a concurrent functional language based on the π-calculus, a basic language for mobile processes. The semantics can be used to extract causal dependencies, as demonstrated by its use to debug a mobile agent system. This work is very promising, and it will be interesting to see how well it scales to mobile code programming languages.

The advantage of the preceding contributions on program analysis or typing [Banâtre et al. 1994; Mizuno and Schmidt 1992; Volpano et al. 1996] is that they lead to mechanical verifications. Their limitation is that they describe individual security analyses for programs without considering their integration in the general context (including the network, operating system, abstract machine, etc.). As a consequence, the properties of the programs verified by the analyses are not linked to a general security policy for the whole system. On the other hand, the approach of Necula and Lee [1996] can be seen as a more ambitious attempt, but it relies on mostly manual proof construction. One challenge for future work is probably to find an appropriate integration of automatic and interactive proof techniques.

APPENDIX. LANGUAGE EXAMPLES

To give the reader a feel of the different languages we present examples for Java, Limbo, O'Caml, and Safe-Tcl. Unfortunately, realistic examples would be much too long for this article. We omit Obliq and Telescript, as we feel there are already sufficient examples of their treatment.

A.1 Java

Figure 1 shows the source of a small applet example illustrating the use of two user interface primitives: buttons and labels.

1–2 The two import statements make all classes from the package applet and awt (abstract windows toolkit) available under their unqualified class name. In this example, we use the class Applet from the package applet and the classes Button and Label from the package awt.

4–6 The Applet class constitutes a framework for applets and all applets are a specialization of the Applet class. The applet in this example, HelloWorld, has two graphics items, a push button and a label. These are declared as the private variables push and lab of class Button and Label, respectively.

8–11 The system initializes applets by calling init. The first thing to do is to call the init defined in the superclass. A new Button instance, initialized to

```
1   import java.applet.*;
2   import java.awt.*;
3
4   public class HelloWorld extends Applet {
5     private Button push;
6     private Label lab;
7
8     public void init() {
9       super.init();
10      add(push = new Button("Push me"));
11      add(lab = new Label("Hello World Applet Demo"));
12    }
13
14    public boolean action(Event event, Object arg) {
15      remove(push);
16      lab.setText("Hello World has ended");
17      return true;
18    }
19  }
```

Figure 1. Source of Java applet HelloWorld.

carry the label "Push me," is created in line 10. This instance is assigned to the local attribute push. This instance is added to the scene[4] through the method add, defined in the class Container (not shown) in the package java.awt.Applet is a specialization of Container.

14–17 Button events are delivered to Applet objects by calling their action method. For this example, it is not necessary to check which button was pressed, as there is only one; thus the two arguments, event and arg, are ignored. The applet responds to the button press by removing the button from the scene (line 15) and changing the message in the label (line 16). remove is a method of Container like add, and setText (line 16) is a method of the class Label.

[4] As in C, assignments are expressions with the value of their left-hand side.

A.2 Limbo

Figure 2 shows a Limbo example that uses a channel to report activations of a button, illustrating communication between Tk and Limbo.

1 The head of the module declares that what follows constitutes the module Hello. Module names begin with uppercase letters.

3– 5 The signature of modules Draw and Tk are included from the files "draw.m" and "tk.m". The tk: Tk statement declares an instance named tk of the module Tk (initialized to nil). Types and constants exported from modules can be accessed directly from the signature of the module, but functions and variables require a module instance. As this example only uses the type Context from module Draw, there is no need to make an instance.

7–9 The signature of this module exports only one function, init. By convention, programs capable of being executed from the

```
1    implement Hello;
2
3    include "draw.m";
4    include "tk.m";
5    tk: Tk;
6
7    Hello: module {
8      init: fn(ctxt: ref Draw->Context, argv: list of string);
9    }
10
11   init(ctxt: ref Draw->Context, argv: list of string) {
12     tk = load Tk Tk->PATH;
13     t := tk->toplevel(ctxt.screen, "");
14     ccmd := chan of string;
15     tk->namechan(t, ccmd, "tcmd");
16     tk->cmd(t, "button .b -text Remove -command {send tcmd bye}");
17     tk->cmd(t, "pack .b");
18     tk->cmd(t, "update");
19     <- ccmd;
20   }
```

Figure 2. Limbo example.

top-level shell have a function called init with the signature: a function taking a graphics context (type Context from module Draw) and a list of string arguments. The graphics context provides a reference to a window system, necessary to create new windows.

11–12 The implementation of the init function starts by loading the implementation of module Tk and creating the instance tk. By convention, each module declaration includes a pathname constant that points to the code for the module; this is the second parameter Tk−>PATH of the load statement.

13 The variable t is declared and initialized to a reference to the top-level window.

14 The string channel ccmd is declared and instantiated. In this example, a message on this channel signals the termination of the application.

15 The namechan call associates the Limbo channel with a Tk string, thus bridging the two languages. Messages sent on tcmd from the Tk language appear on ccmd.

16–18 These three lines are calls to the Tk graphics library to create a button that sends the string "bye" on tcmd (ccmd) upon a mouse press. The pack command places the button .b on the top level window, and update makes it appear on the screen.

19–20 The last thing to do is to wait for a message on the channel. The value received is ignored.

A.3 O'Caml

Figure 3 shows the complete source of an MMM timer applet.

1 The open statement imports all publicly exported identifiers from the module.

2–5 Safestd is a safe subset of the standard library, containing basic primitives, such as

```
1   open Safestd
2   open Safemmm
3   open Safetk
4   open Safeunix
5   open Tk
6
7   let f a b c =
8     let t = Toplevel.create (Mmm.get_global_widget()) [] in
9     Wm.title_set t "Time Web";
10    let l = Label.create t [Text "00:00:00"] in
11    let upd () =
12     let tm = localtime(time()) in
13     let txt n =
14        if n < 10 then "0" ^ string_of_int n
15        else string_of_int n in
16     let tms = txt tm.tm_hour^":"^txt tm.tm_min^":"^txt tm.tm_sec in
17      Label.configure l [Text tms]
18   in
19    let rec tim () =
20      if Winfo.exists l then begin
21        add_timer 1000 tim;
22        upd()
23        end in
24    tim();
25    pack [l][Fill Fill_X]
26
27  let a = Applets.register "f" f
```

Figure 3. O'Caml applet source.

string_of_int in the following. The module Safemmm gives restricted access to internals of the MMM browser. Here we only need it to create a top-level window. Safetk contains the graphics toolkit function widgets.

7 The applet is a top-level function f with three arguments of no relevance for this example.

8–9 Creates a top-level window and sets the title to "Time Web."

10 Applies constructor Text to string "00:00:00," creating an initialized text label 1 (a graphic element). Text is imported from the module Tk.

11–17 Defines a function to format the current local time as a string and update the label 1. The ˆ operator is for string concatenation.

19–23 The tim function updates the text label continuously (with a small pause). Winfo.exists 1 checks that the window is still present (the user can delete it externally using the window manager). If the window has been removed, the applet stops. The function add_timer registers a thunk (a function of type ( ) → Unit) for execution after a given number of milliseconds. The call to add_timer itself returns immediately.

24–25 The updating is started and the label is installed in the top-level window. The list Fill Fill_X are options to pack, making the label take all

```
1   proc ordershirt {} {
2     SafeTcl_sendmessage -to tshirts@nowhere.really \
3           -subject "Shirt request" \
4           -body [SafeTcl_makebody "text/plain" \
5                  [SafeTcl_getline \
6                     "What size t-shirt do you wear?" \
7                     "medium"] ""]
8     exit
9   }
10
11  if {[lsearch $SafeTcl_InterfaceStyle "Tk3.*"] >= 0} {
12      set win [mkwindow]
13      message $win.m -aspect 1000 \
14        -text "Click below if you want a free Bill Clinton t-shirt!"
15      button $win.b -text "Click here for free shirt!" \
16        -command {ordershirt}
17      button $win.b2 -text "Click here to exit without ordering" \
18        -command exit
19      pack append $win $win.m {pady 20} $win.b {pady 20} \
20        $win.b2 {pady 20}
21  } else { ... }
```

Figure 4. SafeTcl applet source.

available space in the window.

27 The final step is to register the applet function, establishing the connection between O'Caml and MMM.

A.4 Safe-Tcl

Figure 4 gives the partial source of a Safe-Tcl example application. In Tcl, the first word of the line is always the name of a command and the rest of the line contains the arguments, separated by spaces. The backslash at the end of the line allows long statements to be broken across several lines. The square brackets group subexpressions. The curly brackets group arguments: everything up to but excluding the matching closing curly brackets is considered one argument. Optional arguments are preceded by a keyword with a leading dash (–), imitating the Unix command-line convention. Variables are accessed with the dollar operator ($). String concatenation is implicit everywhere; for example, $win.m is the contents of the win variable followed immediately by the string ".m".

1–9 We define a function, ordershirt, which queries the user for size information, using the SafeTcl_getline function. The result is wrapped up in the body of a mail by SafeTcl_ makebody and sent to tshirts@nowhere.really (by SafeTcl_ sendmessage).

11 The conditional checks whether it is running in a Tk-capable client.

12 Creates a top-level window and stores it in the variable win. The command set assigns variables to values.

13–18 Equips the window with a leading message and two buttons, one of which activates the previously defined ordershirt upon button press. The built-in commands message and button create a message panel and a button graphic element, respectively.

19–20 The pack command groups the graphics elements together. The pady 20 argument specifies a vertical filling factor.

REFERENCES

ADL-TABATABAI, A., LANGDALE, G., LUCCO, S., AND WAHBE, R. 1996. Efficient and language-independent mobile programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.*

ARNOLD, K., AND GOSLING, J. 1996. *The Java Programming Language.* Sun Microsystems.

BALL, S. 1996. The SurfIt! browser. Available at http://pastime.anu.edu.au/SurfIt.

BANÂTRE, J., BRYCE, C., AND LE MÉTAYER, D. 1994. Compile-time detection of information flow in sequential programs. In *Proceedings of the European Symposium on Research in Computer Security*, No. 875 in *Lecture Notes in Computer Science.* Springer-Verlag, New York.

BELL, D., AND LAPADULA, L. 1973. Secure computer system: Mathematical foundations and model. Tech. Rep. M74–244, MITRE Corp.

BILLON, J.-P. 1996. Java security: Weaknesses and solutions. Available at http://www.dyade.fr/actions/VIP/JS_pap2.html.

BIRRELL, A. D., NELSON, G., OWICKI, S., AND WOBBER, E. 1993. Network objects. In *Proceedings of the 14th Symposium on Operating Systems Principles.*

BJØRNER, D. 1991a. *Software Architectures and Programming Systems Design; Volume I: Specification Principles—the VDM Approach.* Addison-Wesley/ACM Press.

BJØRNER, D. 1991b. *Software Architectures and Programming Systems Design; Volume II: Implementation Principles—the VDM Approach.* Addison-Wesley/ACM Press.

BORENSTEIN, N. S. 1994. Email with a mind of its own. Available at ftp://ftp.fv.com/pub/code/other/safe-tcl.tar.gz, 1994.

BORGIA, R., DEGANO, P., PRIAMI, C., LETH, L., AND THOMSEN, B. 1996. Understanding mobile agents via a non-interleaving semantics for Facile. In *Proceedings of SAS '96, Vol. 1145 of Lecture Notes in Computer Science,* Springer-Verlag, 98–112.

BROCK, S., AND GEORGE, C. W. 1990. The RAISE method manual. Tech. Rep. LACOS/CRI/DOC/3, CRI: Computer Resources International.

CARDELLI, L. 1995. A language with distributed scope. In *Proceedings of the 22nd Symposium on Principles of Programming Languages* (Jan.), ACM Press, New York, 286–297. Available at http://www.research.digital.com/SRC/personal/Luca_Cardelli/Obliq/Obliq.html.

DEAN, D., FELTEN, E. W., AND WALLACH, D. S. 1996. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy* (Oakland, CA).

DENNING, D., AND DENNING, P. 1977. Certification of programs for secure information flow. *Commun. ACM 20,* 7.

DROSSOPOULOU, S., AND EISENBACH, S. 1996. Proving the soundness of the Java type system. Tech. Rep., Imperial College, Oct.

FOX, R. 1996. Internet sabotage. *Commun. ACM 39,* 11 (Nov.), 9.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. The SSL protocol. Available at http://home.netscape.com/eng/ssl3/index.html, March.

GENERAL MAGIC. 1996. The Telescript home page. Available at http://www.genmagic.com/Telescript.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification.* Sun Microsystems.

HANSEN, W. J. 1990. Enhancing documents with embedded programs: How Ness extends insets in the Andrew toolkit. In *Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages*, (New Orleans, March) IEEE Computer Society Press, Los Alamitos, CA, 23–32.

HUITEMA, C. 1996. *IPv6: The New Internet Protocol.* Prentice Hall, Englewood Cliffs, NJ.

JAVASOFT. 1996. JavaSoft products. Available at http://www.javasoft.com/nav/read/products.html.

KNABE, F. 1996. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages.* (Stockholm, Sweden, June) No. 1192 in *Lecture Notes in Computer Science*, Springer-Verlag, New York.

LANDWEHR, C. E. 1981. Formal models for computer security. *ACM Comput. Surv. 13,* 3, 247–278.

LANO, K. 1996. *The B Language and Method.* Springer-Verlag.

LEROY, X. 1997. Objective Caml. Available at http://pauillac.inria.fr/ocaml/.

LUCENT TECHNOLOGIES. 1996. The Inferno home page. Available at http://inferno.bell-labs.com/inferno/index.html.

MALENFANT, J. 1995. On the semantic diversity of delegation-based programming languages. In *Proceedings of OOPSLA '95.*

MCLEAN, J. 1994. Security models. In *Encyclopedia of Software Engineering*, J. Mariniak, Ed., John Wiley & Sons, New York.

MIZUNO, M., AND SCHMIDT, D. A. 1992. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects Comput. 4,* 6A, 727–754.

NECULA, G. C., AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*. Available at http://fox-net.cs.cmu.edu/people/petel/papers/osdi.ps.

NETSCAPE. 1997. JavaScript language specification. Available at http://developer.netscape.com/library/documentation/index.html.

NIELSON, H. R., AND NIELSON, F. 1992. *Semantics with Applications, a Formal Introduction.* Wiley Professional Computing. John Wiley and Sons, New York.

OUSTERHOUT, J. K. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA.

PIKE, R. 1997. Private communication.

RESCORLA, E., AND SCHIFFMAN, A. 1996. The secure hypertext transfer protocol. Available at ftp://ds.internic.net/internet-drafts/draft-ietf-wts-shttp-03.txt, July.

ROUAIX, F. 1996a. MMM browser home page. Available at http://pauillac.inria.fr/~rouaix/mmm/.

ROUAIX, F. 1996b. A Web navigator with applets in Caml. In *Proceedings of the Fifth International World-Wide Web Conference* (Paris, May). Elsevier Science B. V.

RUSSELL, D., AND GANGEMI, G. T., SR. 1991. *Computer Security Basics.* O'Reilly & Associates, Inc., Sebastopol, CA.

SCHMIDT, D. A. 1986. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Needham Heights, MA.

SPIVEY, J. M. 1988. *Understanding Z: A Specification Language and Its Formal Semantics.* Cambridge University Press, New York.

VOLPANO, D., SMITH, G., AND IRVINE, C. 1996. A sound type system for secure flow analysis. *J. Comput. Sec.* 4(3).

Received December 1996; accepted May 1997