SAFKASI[1]: A Security Mechanism for Language-based Systems

Dan S. Wallach
Rice University
Andrew W. Appel and Edward W. Felten
Princeton University

In order to run untrusted code in the same process as trusted code, there must be a mechanism to allow dangerous calls to determine if their caller is authorized to exercise the privilege of using the dangerous routine. Java systems have adopted a technique called stack inspection to address this concern. But its original definition, in terms of searching stack frames, had an unclear relationship to the actual achievement of security, over-constrained the implementation of a Java system, limited many desirable optimizations such as method inlining and tail recursion, and generally interfered with interprocedural optimization. We present a new semantics for stack inspection based on a belief logic and its implementation using the calculus of *security-passing style* which addresses the concerns of traditional stack inspection. With security-passing style, we can efficiently represent the security context for any method activation, and we can build a new implementation strictly by rewriting the Java bytecodes before they are loaded by the system. No changes to the JVM or bytecode semantics are necessary. With a combination of static analysis and runtime optimizations, our prototype implementation shows reasonable performance (although traditional stack inspection is still faster), and is easier to consider for languages beyond Java.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: object-oriented programming; D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.3.2 [**Programming Languages**]: Language Classifications—*Java, object-oriented languages*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls, authentication*

General Terms: Languages, Security, Design

Additional Key Words and Phrases: Java, Internet, WWW, applets, access control, stack inspection, security-passing style

---

[1]The Security Architecture Formerly Known as Stack Inspection

---

Some of the material here has appeared earlier in the *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998, Oakland, California.
Address: Department of Computer Science, Rice University, P.O. 1892, MS 132, Houston, TX 77251-1892

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

1. INTRODUCTION

Java [Gosling et al. 1996] and a number of other recent systems (including Microsoft's C# [Wille 2000]) have considered the problem of running untrusted and trusted code together. To mediate access to potentially dangerous system resources, such as the file system or network, such systems need a security architecture that can flexibly grant different privileges to code having different levels of trust. Several architectures have been proposed [Wallach et al. 1997] but Java vendors have all chosen to use a fairly new technique called stack inspection [Gong and Schemers 1998; Netscape Communications Corporation 1997; Microsoft Corporation 1997].

Stack inspection is an algorithm for preventing untrusted code from using sensitive system resources. Before a dangerous operation proceeds, a call is made to the "security manager," which implements a reference monitor. The security manager will consider, in sequence, the *principals* that "own" each stack frame. A principal is either the Web site from which the code was loaded or a signer who applied a digital signature to the code. If a stack frame's principal is found to be unprivileged for the operation in question, permission is denied and the operation fails (see figure 1).

In some cases, trusted code will use a dangerous resource, such as general access to the file system and network, and export a safe service, such as the ability to load a Web URL using a file cache. To make this possible, an additional mechanism allows trusted code to "enable its privileges," serving as an explicit assertion to take responsibility for future dangerous actions. This assertion is generally recorded as a mark upon the caller's stack frame, and is later recognized by the security manager. Stack inspection has numerous software engineering and security benefits, including higher assurance that software bugs will not lead to security violations. We discuss the space of possible security architectures and their relative benefits in Wallach, Balfanz, Dean, and Felten [1997].

In a stack-inspection system, each method (or procedure) is owned by a *principal*, and each principal has a finite set of *privileges*. The primitive operations in a stack-inspection system, and their explanation are:

BeginPrivilege() Enable privileges. Implemented by Netscape and Microsoft by marking the current stack frame as privileged. Implemented by Sun by passing an object to `DoPrivilged()` which is involved in a new and privileged stack frame.

CheckPrivilege(T) Check whether privileges for a given target T are enabled. CheckPrivilege is implemented by searching stack frames from the current one. If a marked frame is found whose owner possesses the privilege to use T, CheckPrivilege is successful and the search terminates. If a frame is found whose owner is not authorized to use T, CheckPrivilege raises an exception. If any other frame is found, the search continues. If the search completes without finding an unprivileged stack frame, the requested operation is either allowed (with Microsoft and Sun) or denied (with Netscape). The exact algorithm appears in figure1.

The exact syntax varies across the implementations of Netscape, Microsoft, and Sun but all behave similarly enough that the differences are unimportant to this discussion.

The rationale for stack inspection is that if an unprivileged method owned by Alice calls the `deleteFile()` method, the CheckPrivilege done in `deleteFile` will fail when it encounters Alice's frame. But suppose Alice calls a system-owned quicksort routine and tries to cheat by passing `deleteFile` (or an object containing that method) as the "compare" function? We do not want `deleteFile` to believe it was authorized by the

```
CheckPrivilege (target) {
    // loop, newest to oldest stack frame
    foreach stackFrame {
        if (local policy forbids access to target by class executing in stackFrame)
            throw ForbiddenException;

        if (stackFrame has enabled privilege for target)
            return; // allow access
    }

    // if we reached here, we fell off the end of the stack
    if (Netscape 4.0)
        throw ForbiddenException;
    if (Microsoft IE 4.0 || Sun Java 2.0)
        return; // allow access
}
```

Fig. 1. Java's stack walking algorithm.

(extremely privileged) system principal just because it was called directly by a system-owned method! In this case, CheckPrivilege will examine the stack frame for quicksort, find that it is neither unprivileged nor enabled, and continue on to find the unprivileged Alice frame, which will cause a failure of CheckPrivilege. This class of attack is sometimes called the *confused deputy problem* [Hardy 1988], where an attacker attempts to take advantage of a subroutine that has more privileges than it needs.

On the other hand, if Alice calls a `LoadURL` method owned by system, which executes BeginPrivilege and then calls another system routine which in turn calls `deleteFile`, the CheckPrivilege call will succeed.

Stack inspection is a technique to help the Java system satisfy the *principle of least privilege* [Saltzer and Schroeder 1975]. With stack inspection, the Java system may operate with less than its full privileges active at all times and thus exposure to attacks is reduced. This proved extremely useful in Netscape 3.0 [Roskind 1996]. An additional benefit of requiring explicit calls to enable privileges was that these calls could be quickly identified with text searching tools such as `grep` and then subjected to code auditing. With limited time to audit a large code base, this technique allows an audit to focus its efforts on code that will effect the security of the system.

Stack inspection, as described above, has three weaknesses:

(1) It is not clear whether this mechanism achieves real security, or how to reason about the security that it might achieve.

(2) The operational definition of stack inspection overconstrains its implementation. An implementation would find it difficult to collapse stack frames by inlining or tail-recursion elimination, since these optimizations would change the state visible to the security system. Any interprocedural analysis would have to take into account these special side effects to the stack frame, and their observation by callees arbitrarily far up the stack. This problem is essentially similar to the problem of supporting program debugging in the presence of optimized code [Hennessy 1982; Tolmach and Appel 1990].

(3) The high cost of the CheckPrivilege operation discourages its use. For example, the

standard IO library calls CheckPrivilege only on opening a file, not on each access to the resulting file descriptor. We conjecture that this design decision was made because CheckPrivilege was expensive.

We have solved all three of these problems:

(1) In sections 3, we show that stack inspection can be explained using a simple belief logic (designed by Abadi, Burrows, Lampson and Plotkin [Abadi et al. 1993] and summarized in section 2).

(2) In sections 4 and 5, we describe a new semantics of stack inspection that we call "security-passing style" [Wallach and Felten 1998], which we prove to be faithful to the original stack inspection model. By implementing this semantics directly, we can express all security operations in plain Java (or any underlying language with procedure calls). Not only does this avoid interference with program analyses and optimizations, but standard dataflow-based compiler optimizations now help optimize the security operations for free.

(3) Each of the operations in our new implementation can be performed in $O(1)$ time. Still, some of these operations are relatively expensive. Section 6 describes a number of optimizations based on static analysis of the program. In many cases, it is possible to identify and remove security operations without changing the semantics of the security system.

In building our system, described in section 7, we constrain ourselves to make no changes to the underlying Java virtual machine [Lindholm and Yellin 1996]. We operate strictly by rewriting Java classes at load time, so the just-in-time compiler needs to know nothing about the transformation we apply to the code. Our implementation works within the context of any standard JVM/JIT, and we show measurements of its performance with a current compiler.

2. ACCESS CONTROL LOGIC

In order to gain a more sophisticated understanding of stack inspection, we found it necessary to build a *model* of the system. A good model would hide many of the details of the system and allow us to reason about it. In particular, we would like the model to capture the "security state" of the system at any time and let us express transitions from this state as mathematical operations.

There are no hard and fast rules of how one should model a system formally. Instead, the most expedient path is to find a formal model of a similar system and adapt it to stack inspection. The system that we decided to borrow from was originally used to describe authentication and access control in the Taos operating system [Lampson et al. 1992; Wobber et al. 1994]. In Taos, the operating system maintains information about every channel between processes on the same machine and across the network. When a process receives a request, the process may ask the system to identify who has connected to it. Because a channel may pass through multiple points of trust (the local operating system, the network, the remote operating system, etc.), the system explicitly puts these into the principal, creating a *compound principal*. Taos actually included a theorem prover inside the system which could, given these compound principals and a security policy, both expressed in the same formal logic, generate proofs of whether a given request is authorized to occur. The logic, a relatively simple propositional or modal logic with no negation of statements (and with certain restrictions on the form of statements), allows the theorem prover to run fast enough to not dramatically impact system performance. We decided to adopt this logic, originally specified by Abadi, Burrows, Lampson, and Plotkin [Abadi et al. 1993] (hereafter, ABLP logic), to model stack inspection.

2.1 ABLP Logic

Stack inspection can be modeled using a subset of ABLP. This section will describe the subset and give a general flavor for how it can be used.

The logic is based on a few simple concepts: principals, conjunctions of principals, targets, statements, quotation, and authority.

—A *principal* is a person, organization or any other entity that may have the right to take actions or authorize actions. In addition, entities such as programs and cryptographic keys are often modeled as principals.

—A *target* represents a resource that we wish to protect. Loosely speaking, a target is something to which we might like to attach an access control list. (Targets are traditionally known as "objects" in the literature, but this can be confusing when talking about an object-oriented language.)

—A *statement* is any kind of utterance a principal can emit. Some statements are made explicitly by a principal, and some are made implicitly as a side-effect of actions the principal takes. In other words, we interpret P **says** s as meaning that we can act as if the principal P supports the statement s. Note that saying something does not make it true; a speaker could make an inaccurate statement carelessly or maliciously. The logic supports the informal notion that we should place faith in a statement only if we trust the speaker and it is the kind of statement that the speaker has the authority to make. Thus, if a speaker makes an inaccurate statement, we will not believe the statement. Also, speakers cannot make statements that lead to a logical contradiction (*e.g.,* $A \supset \neg A$) because negation is not allowed in ABLP.
The most common type of statement we will use looks like P **says** $Ok(T)$ where P is a principal and T is a target; this statement means that P is authorizing access to the target T. By saying an action is "*Ok*" the speaker is saying the action should be allowed in the current context but is not specifically ordering that the action take place.

—The logic supports *conjunctions of principals*. Specifically, saying $(A \wedge B)$ **says** s is the same as saying both A **says** s and B **says** s.

—*Quotation* allows a principal to make a statement about what another principal says. The notation $A \mid B$ **says** s, which we pronounce "A quoting B says s," is equivalent to A **says** $(B$ **says** $s)$. As with any statement, we must consider whether A's utterance might be incorrect, and our degree of faith in s will depend on our beliefs about A and B. When A quotes B, we have no guarantee that B ever actually said anything.

—We grant *authority* to a principal by allowing that principal to speak for another principal who has power to do something. The statement $A \Rightarrow B$, pronounced "A speaks for B," means that if A makes a statement, we can assume that B supports the same statement. If $A \Rightarrow B$, then A has at least as much authority as B. Note that the $\Rightarrow$-operator can be used to represent group membership: if P is a member of the group G, we can say $P \Rightarrow G$, meaning that P can exercise the rights granted to G.
When proving theorems, the $A \Rightarrow B$ means occurrences of A can be replaced with $A \wedge B$. Thus, when we hear a statement from A, we can act as if it were spoken jointly by A and

B together.

3. MAPPING JAVA TO ABLP

We will now describe a mapping from the stack, the privilege calls, and the stack inspection algorithm into ABLP logic.

3.1 Principals

In Java, code is digitally signed with a private key, then shipped to the virtual machine where it will run. If K_{Signer} is the public key of *Signer*, the public-key infrastructure can generate a proof[2] of the statement

$$K_{Signer} \Rightarrow Signer. \tag{1}$$

Signer's digital signature on the code *Code* is interpreted as

$$K_{Signer}\ \textbf{says}\ (Code \Rightarrow K_{Signer}). \tag{2}$$

Using equations 1 and 19 (see appendix A.2), this implies that

$$Code \Rightarrow Signer. \tag{3}$$

When *Code* is invoked, it generates a stack frame *Frame*. The virtual machine assumes that the frame speaks for the code it is executing:

$$Frame \Rightarrow Code. \tag{4}$$

The transitivity of $\Rightarrow$ (which can be derived from equation 18) then implies

$$Frame \Rightarrow Signer. \tag{5}$$

We define Φ to be the set of all such valid $Frame \Rightarrow Signer$ statements. We call Φ the *frame credentials*.

Note also that code can be signed by more than one principal. In this case, the code and its stack frames speak for all of the signers. Likewise, a "signer" can also be a "code base" (*e.g.*, a local directory, a Web site, etc.) or the combination of a digital signature and a code base. To simplify the discussion, all of our examples will use single signers, but the theory can support multiple signers or signers with code bases without difficulty.

3.2 Targets

The resources we wish to protect are called *targets*. For each target, we create a dummy principal whose name is identical to that of the target. These dummy principals do not make any statements themselves, but various principals may speak for them.

For each target T, the statement $Ok(T)$ means that access to T should be allowed in the present context. The axiom

$$(T\ \textbf{says}\ Ok(T)) \supset Ok(T) \tag{6}$$

says that T can allow access to itself.

While targets may be defined in relation to services offered directly within Java (*e.g.* access to an in-memory database), targets are most commonly defined in relation to services

[2]Throughout this paper we assume that sound cryptographic protocols are used, and we ignore the extremely unlikely possibility that an adversary will successfully guess or otherwise acquire a private key.

offered by the operating system underlying the Java Virtual Machine (JVM). From the operating system's point of view, the JVM is a single process and all system calls coming from the JVM are performed under the authority of the JVM's principal (often the user running the JVM). The JVM's responsibility, then, is to allow a system call only when there is justification for issuing that system call under the JVM's authority. Our model will support this intuition by requiring the JVM to prove in ABLP logic that each system call has been authorized by a suitable principal.

Even if a JVM were running directly on the hardware without an operating system (*e.g.*, JavaOS [Saulpaugh et al. 1999]), control flow for system operations would eventually reach a device-driver that likely knows nothing about privileged vs. unprivileged operation. At some point, control must necessarily have crossed a "red line" where sufficient information was still available to perform a security check. Security checks would naturally occur, and targets would naturally be defined at these boundaries.

3.3 Setting Policy

We use a standard access matrix [Lampson 1971], implemented with with hashtables to achieve compact storage, to keep track of which principals have permission to access which targets. If *VM* is a Java virtual machine, we define $\mathcal{A}_{VM}$ to be a set of statements of the form $P \Rightarrow T$ where P is a principal and T is a target. If $(P \Rightarrow T) \in \mathcal{A}_{VM}$, this means that the local policy in *VM* allows P to access T. We call $\mathcal{A}_{VM}$ the *access credentials* for the virtual machine *VM*.

3.4 Stacks

When a Java program is executing, we treat each stack frame as a principal. At any point in time, a stack frame F has a set of statements that it believes. We refer to this as the *security context* of F and write it $\mathcal{S}_F$. We now describe where the security context comes from.

3.4.1 *Starting a Program.* When a program starts, we need to set the security context of the initial stack frame, $\mathcal{S}_{F_0}$. In the Netscape model, $\mathcal{S}_{F_0} = \{\}$. In the Sun and Microsoft models, $\mathcal{S}_{F_0} = \{Ok(T) \mid T \in \mathit{Targets}\}$. These correspond to Netscape's initial unprivileged state and Sun and Microsoft's initial privileged state.

3.4.2 *Enabling Privileges.* If a stack frame F calls BeginPrivilege(T) for some target T, it is really saying it *authorizes* access to the target. We can represent this simply by adding $Ok(T)$ to $\mathcal{S}_F$.

3.4.3 *Calling a Procedure.* When a stack frame F makes a procedure call, this creates a new stack frame G. As a side-effect of the creation of G, F tells G the statements in F's security context. Thus, when F tells G a statement S, the statement F **says** S is added to $\mathcal{S}_G$.

3.5 Checking Privileges

Before making a system call or otherwise invoking a dangerous operation, the Java virtual machine calls CheckPrivilege() to make sure that the requested operation is authorized. CheckPrivilege(T) returns true if the statement $Ok(T)$ can be derived from Φ (the frame credentials), $\mathcal{A}_{VM}$ (the access control matrix), and $\mathcal{S}_F$ (the security context of the frame that called CheckPrivilege()).

We define *VM(F)* to be the virtual machine in which a given frame F is running. Next, we can define

$$\mathcal{E}_F \equiv (\Phi \cup \mathcal{A}_{VM(F)} \cup \mathcal{S}_F). \tag{7}$$

We call $\mathcal{E}_F$ the *environment* of the frame F.

The goal of CheckPrivilege(T) is to determine, for the frame F invoking it, whether $\mathcal{E}_F \supset Ok(T)$. While such questions are generally undecidable in ABLP logic, we now present an efficient decision procedure that gives the correct answer for our subset of the logic. CheckPrivilege() implements that decision procedure.

The decision procedure **check** used by CheckPrivilege() takes, as arguments, an environment $\mathcal{E}_F$ and a target T. **check**$(T, \mathcal{E}_F)$ examines the statements in $\mathcal{E}_F$ and divides them into three classes.

—Class 1 statements have the form $Ok(U)$, where U is a target.

—Class 2 statements have the form $P \Rightarrow Q$, where P and Q are atomic principals.

—Class 3 statements have the form

$$F_1 \mid F_2 \mid \cdots \mid F_k \textbf{ says } Ok(U),$$

where F_i is an atomic principal for all i, $k \geq 1$, and U is a target.

The decision procedure next examines all Class 1 statements. If any of them is equal to $Ok(T)$, the decision procedure terminates and returns *true*.

Next, the decision procedure uses all of the Class 2 statements to construct a directed graph which we will call the speaks-for graph of $\mathcal{E}_F$. This graph has an edge (A, B) if and only if there is a Class 2 statement $A \Rightarrow B$.

Next, the decision procedure examines the Class 3 statements one at a time. When examining the statement $F_1 \mid F_2 \mid \cdots \mid F_k$ **says** $Ok(U)$, the decision procedure terminates and returns *true* if both

—for all $i \in [1, k]$, there is a path from F_i to T in the speaks-for graph, and

—$U = T$.

If the decision procedure examines all of the Class 3 statements without success, it terminates and returns *false*.

THEOREM 1. **Termination.** *The* **check** *decision procedure always terminates.*

Proof: The result follows directly from the fact that $\mathcal{E}_F$ has finite cardinality; there are a finite number of principals that the system knows about, a finite number of stack frames that must be considered, and a security policy of finite length. This implies that each loop in the algorithm has a bounded number of iterations; and clearly the amount of work done in each iteration is bounded. □

In fact, the decision procedure runs quite efficiently. We can separately analyze the runtime complexity and space complexity of each phase, as presented above. If there are N rules in the access control matrix $\mathcal{A}_{VM(F)}$ and a stack depth of D (*i.e.,* Φ has at most D elements), the cost of computing the transitive closure of the graph will be, at worst, $O((N+D)^2)$ and consume $O((N+D)^2)$ space. Then, if there are k statements in $\mathcal{S}_F$ (each of which can have at most D principals in its quoting chain), the cost of checking all statements will be $O(kD)$. The total cost of the decision procedure is thus $O((N+D)^2 + kD)$.

In practice, the transitive closure of the access control policy can be pre-computed (subject to the caveats in section 3.6.1) and the $O(kD)$ complexity of the security context can be lowered as well. In section 4.1, we describe optimizations that allow the decision procedure to execute in constant time.

THEOREM 2. **Soundness.** *If the* **check** *decision procedure returns* true *when invoked in stack frame* F, *then there exists a proof in ABLP logic that* $\mathcal{E}_F \supset Ok(T)$.

LEMMA 1. *If there is a path from* A *to* B *in the speaks-for graph of* $\mathcal{E}_F$, *then* $\mathcal{E}_F \supset (\mathrm{A} \Rightarrow \mathrm{B})$.

Proof: By assumption, there is a path

$$(A,\ v_1,\ v_2,\ \ldots,\ v_k,\ B)$$

in the speaks-for graph of $\mathcal{E}_F$. In order for this path to exist, we know that the statements

$$A \Rightarrow v_1,$$

$$v_i \Rightarrow v_{i+1} \text{ for all } i \in [1, k-1],$$

and

$$v_k \Rightarrow B$$

are all members of $\mathcal{E}_F$. Since $\Rightarrow$ is transitive, this implies that

$$\mathcal{E}_F \supset A \Rightarrow B.$$

Proof of Theorem 2: There are two cases in which the **check** decision procedure can return *true*.

(1) The decision procedure returns *true* while it is iterating over the Class 1 statements. This occurs when the decision procedure finds the statement $Ok(T) \in \mathcal{E}_F$. In this case, $Ok(T)$ follows trivially from $\mathcal{E}_F$.

(2) The decision procedure returns *true* while it is iterating over the Class 2 statements. In this case we know that the decision procedure found a Class 2 statement of the form

$$P_1 \mid P_2 \mid \cdots \mid P_k \textbf{ says } Ok(T),$$

where for all $i \in [1, k]$ there is path from P_i to T in the speaks-for graph of $\mathcal{E}_F$. It follows from Lemma 1 that for all $i \in [1, k]$, $P_i \Rightarrow T$. It follows that

$$\mathcal{E}_F \supset (T \mid T \mid \cdots \mid T \textbf{ says } Ok(T)). \tag{8}$$

Applying equation 6 repeatedly, we can directly derive $\mathcal{E}_F \supset Ok(T)$. □

CONJECTURE 1. **Completeness.** *If the* **check** *decision procedure returns false when invoked in stack frame* F, *then there is no proof in ABLP logic of the statement* $\mathcal{E}_F \supset Ok(T)$.

Although we believe this conjecture to be true, we do not presently have a complete proof. If the conjecture is false, then some legitimate access may be denied. However, as a result of theorem 2, no access will be improperly granted.

If the conjecture is true, then Java stack inspection, our access control decision procedure, and proving statements in our subset of ABLP logic are all mutually equivalent.

THEOREM 3. **Equivalence to Stack Inspection.** *The* **check** *decision procedure is equivalent to the Java stack inspection algorithm of figure 1.*

Proof: The Java stack inspection algorithm (Figure 1) itself does not have a formal definition. However, we can treat the evolution of the system inductively and focus on the BeginPrivilege() and CheckPrivilege() primitives.

We wish to prove that given a Java stack S and its ABLP-modeled equivalent $M(S)$, and for all targets T, CheckPrivilege$(T,S) \equiv$ **check**$(T,M(S))$.

Our induction is over the number of *steps* taken, where a step is either a procedure call or a BeginPrivilege() operation. Steps are defined as operations on environments. For clarity, we ignore procedure return operations; our proof can easily be extended to accommodate them.

We also assume Netscape semantics. A simple adjustment to the base case can be used to prove equivalence between the decision procedure and the Sun/Microsoft semantics.

Base case: In the base case, no steps have been taken. In this case, the stack inspection system has a single stack frame with no privilege annotation; in the ABLP model, the stack frame's security context is empty. In this base case, CheckPrivilege(T,S_0) and **check**$(T,M(S_0))$ will both return false.

Inductive step: We assume that N steps have been taken ($N \geq 0$) and we are in a situation where both CheckPrivilege(T,S) and **check**$(T,M(S))$ would yield the same result. Now there are two cases: **BeginPrivilege(T) step:** In the stack inspection system, this adds an *enabled-privilege*(T) annotation on the current stack frame. In the ABLP model, it adds *Ok*(T) to the current security context (a part of $M(S)$).

If this BeginPrivilege() call is followed by a call to CheckPrivilege(T), the Java stack inspection algorithm will succeed because the *enabled-privilege*(T) flag is immediately discovered. Likewise, a call to **check**$(T,M(S))$ will succeed because *Ok*(T) is found in $M(S)$, *Ok*(T) is what **check** is trying to prove.

If this BeginPrivilege() call is followed by a call to CheckPrivilege(U) with $U \neq T$, the new stack annotation or statement will be irrelevant to the result of either CheckPrivilege() or **check**, so we fall back on the inductive hypothesis to show that both systems give the same result.

Procedure call step: Let P be the principal of the procedure that is called. In the stack inspection system, this adds to the stack an unannotated stack frame belonging to P. In the ABLP system, it prepends "P **says**" to the front of every statement in the current security context.

When CheckPrivilege(T) is called, two things occur. First, the call is treated as a normal procedure call, with the caller's principal being prepended to the statements in the security context. Then, there are two sub-cases.

—***P* is not trusted for *T*.** In the stack inspection case, CheckPrivilege(T) will fail because the current frame is not trusted to access T. In the ABLP case, the **check** will deny access because every statement starts with "P **says**" and P does not speak for T.

—***P* is trusted for *T*.** In the stack inspection case, the stack search will ignore the current frame and proceed to the next frame on the stack. In the ABLP case, since $P \Rightarrow T$, the "P **says**" on the front of every statement has no effect. Thus both systems give the same answer they would have given before the last step. By the inductive hypothesis, both systems thus give the same result. □

3.6 Extensions to the Model

There are a number of cases in which Java implementations differ from the model we have described. These are minor differences with no effect on the strength of the model.

3.6.1 *Groups.* It is natural to extend the model by allowing the definition of groups. In ABLP logic, a group is represented as a principal, and membership in the group is represented by saying the member speaks for the group. Deployed Java systems use groups in several ways to simplify the process of defining policy.

The Microsoft system defines "security zones" that are groups of principals. A user or administrator can divide the principals into groups with names like "local", "intranet", and "internet", and then define policies on a per-group basis.

Netscape defines "macro targets" that are groups of targets. A typical macro target might be called "typical game privileges." This macro target would speak for those privileges that network games typically need.

The Sun system has a general notion of targets in which one target can imply another. In fact, each target is required to define an `implies()` procedure, which can be used to ask the target whether it signifies a superset of the privileges associated with another target. This can be handled with a simple extension to the model.

Unfortunately, Sun has not clearly defined whether the `implies()` relationships are transitive, and if they are, whether this information may be used to optimize security queries. One useful optimization might be to compute the transitive closure of the `implies()` graph, which would then allow for constant-time queries. However, if a target is allowed to change its mind about what other targets it implies, the security system might be forced to reevaluate the `implies()` relationships on every security query.

3.6.2 *Threads.* Java is a multi-threaded language, meaning there can be multiple threads of control, and hence multiple stacks can exist concurrently.

When a new thread is created in Netscape's system, the first frame on the new stack begins with all privileges disabled. This is modelled as the new stack beginning with an empty security context.

In Sun and Microsoft's systems, the first frame on a new stack inherits the security context of the previous thread at the time the new thread is created. This is done by doing a full stack inspection operation, capturing the state, and storing it in thread-local storage. Later, when a privilege is checked, the normal stack inspection algorithm will first consider stack frames of the current thread then will continue through the saved state from the parent thread. This is modelled as the new stack being passed a reference to its parents security context in precisely the same fashion as when a normal procedure call occurs.

3.6.3 *Enabling a Privilege.* The model of BeginPrivilege() in section 3.4.2 differs somewhat from the Netscape implementation of stack inspection, where a stack frame F cannot successfully call BeginPrivilege(T) unless the local access credentials include $F \Rightarrow T$. The restriction imposed by Netscape is related to their user interface and is not necessary in our formulation, since the statement F **says** $Ok(T)$ is ineffectual unless $F \Rightarrow T$. Sun Java 2.0's implementation is closer to our model.

3.6.4 *Disabling a Privilege.* Netscape supports a primitive to *disable* a privilege. In the stack inspection model, this is implemented by writing an appropriate flag to the stack frame that requests a privilege to be disabled. In the normal stack inspection step, disabled privileges are checked explicitly before enabled privileges. In the event a disabled privilege is discovered, the search immediately terminates.

In our standard model, disabling a privilege for a target T could be implemented as dropping any statements of the form $x \Rightarrow T$ from the current belief set. However, privilege disabling does not mix well with grouping extensions (see section 3.6.1). In particular, if a group of either principals or targets was enabled and a different group was disabled, the system would either need to perform set-subtraction, or would need to retain both positive and negative statements, as well as the order in which to evaluate them. In addition to becoming logically more complex, this would inhibit many of the optimizations described in the remainder of this paper.

In practice, programmers use the *disable* operation in combination with an *enable* operation to bracket a dangerous operation. Sun captures this notion with their `DoPrivileged()` operation, which is passed, as an argument, an object to be invoked with higher privilege. This is equivalent to that object invoking `BeginPrivilege()` when it starts execution. These privileges naturally disappear when the privileged method returns.

3.6.5 *Frame Credentials.* Java implementations do not treat stack frames or their code as separate principals. Instead, they track only the public key that signed the code and call this the frame's principal. As we saw in section 3.1, for any stack frame, we can prove the stack frame speaks for the public key that signed the code. In practice, neither the stack frame nor the code speaks for any principal except the public key. Likewise, access control policies are represented directly in terms of the public keys, so there is no need to separately track the principal for which the public key speaks. As a result, the Java implementations say the principal of any given stack frame is exactly the public key that signed that frame's code. This means that Java implementations need not have an internal notion of the frame credentials described here.

4. IMPROVED IMPLEMENTATION

In addition to improving our understanding of stack inspection, our model and decision procedure can help us find more efficient implementations of stack inspection. We improve the performance in two ways. First, we show that the evolution of security contexts can be represented by a deterministic pushdown automaton; this opens up a variety of efficient implementation techniques. Second, we describe *security-passing style*, an efficient and convenient integration of the pushdown automaton with the state of the program.

4.1 Security Contexts and Automata

We can simplify the representation of security contexts by making two observations about our decision procedure.

(1) Interchanging the positions of two principals in any quoting chain does not affect the outcome of the decision procedure.

(2) If P is an atomic principal, replacing $P \mid P$ by P in any statement does not affect the result of the decision procedure.

Both observations are easily proven, since they follow directly from the structure of the decision procedure.

We also use the observation in section 3.6.5 that we need not consider frame credentials, but need only consider the signer of a given stack frame. This means that multiple stack frames corresponding to the same signature will be considered to have the same principal.

It then follows that without affecting the result of the decision procedure we can rewrite each statement in the security context into a canonical form in which each atomic principal appears at most once, and the atomic principals appear in some canonical order. After this transformation, we can discard any duplicate statements from the security context.

Since the set of atomic principals is finite, the set of targets is finite, and no principal or target may be mentioned more than once in a canonical-form statement, there is therefore a finite set of possible canonical-form statements. It then follows that only a finite number of canonical-form security contexts may exist.

While the number of possible security contexts can grow exponentially in the number of principals and targets, it is nonetheless finite. Therefore, we can represent the evolution of a stack frame's security contexts by a finite automaton, where each state in the automaton corresponds to a security context. Since stack frames are created and destroyed in LIFO order, the execution of a thread can be represented by a finite pushdown automaton, where calling a procedure corresponds to a `push` operation (and a state transition), returning from a procedure corresponds to a `pop` operation, and BeginPrivilege() corresponds to transitions on the finite state graph.

Representing the system as an automaton has several advantages. It allows us to use analysis tools such as model checkers to derive properties of particular policies. It also admits a variety of efficient implementation techniques such as lazy construction of the state set and the use of advanced data structures.

Furthermore, the results of a security check can be stored along with the security contexts. So in cases where the same security check may be made numerous times (such as when one program opens a multitude of files), only the first check would require invoking the decision procedure. Subsequent security checks could consult a local cache and execute in constant time.

One concern is that, because the space of all possible security contexts is exponential in the number of principals and targets, the amount of memory needed will be similarly exponential. This concern is addressed by noting that very few of these security contexts will ever be used. A lazy implementation, one which only allocates memory for security contexts as they are needed, would only allocate memory proportional to the complexity of its security needs. Thus, if the execution of a program only uses a handful of distinct security contexts, only those few contexts will be allocated. Conversely, if a program is truly exponential in its security complexity (i.e., the number of unique security contexts used during the lifetime of a program is exponential) , it would need to run for an exponential amount of time in order to cause the full security context space to be instantiated. Such degenerate cases are unlikely to occur in practice.

4.2 Security-Passing Style

The implementation discussed thus far has the disadvantage that security state is tracked separately from the rest of the program's state. This means that there are two subsystems (the security subsystem and the code execution subsystem) with separate semantics and separate implementations of pushdown stacks coexisting in the same Java Virtual Machine (JVM). We can improve this situation by implementing the security mechanisms in terms of the existing JVM mechanisms.

We do this by adding an extra, implicit argument to every procedure. The extra argument is a pointer into the finite-state space of the automaton. This eliminates the need to have a separate pushdown stack for security contexts or maintain stack annotations on the existing run-time stack. We dub this approach *security-passing style*, by analogy to continuation-passing style [Steele 1978], a transformation technique used by some compilers that also replaces an explicit pushdown stack with implicitly-passed procedure arguments. An implementation of security-passing style is presented in section 7.

The main advantage of security-passing style is that once a program has been rewritten into SPS, it no longer needs any special security functionality from the JVM. The rewritten program consists of ordinary Java bytecode that can be executed by any JVM, even one that knows nothing about stack inspection. This has many advantages, including portability and efficiency. The main performance benefit is that the JVM can use standard compiler optimizations such as dead-code elimination and constant propagation to remove unused security tracking code, or inlining and tail-recursion elimination to reduce procedure call overhead.

Another advantage of security-passing style is that it lets us express the stack inspection model within the existing semantics of the Java language, rather than requiring an additional and possibly incompatible definition for the semantics of the security mechanisms. Security-passing style also lets us more easily transplant the stack inspection idea into other language and systems.

5. THE SECURITY-PASSING STYLE TRANSFORMATION

This section describes the design of the security-passing style transformation. Note that our SPS model is simpler than than the stack inspection model employed by the various Java vendors. Our model only supports BeginPrivilege(), CheckPrivilege(), and function calls. Furthermore, in the simplified model, one may only enable privileges for a specific root target T_{root}, where $\forall targets T_x : T_{root} \Rightarrow T_x$. As a shorthand, we write BeginPrivilege() with no target argument and speak of *Ok*() with no target. These restrictions could all be lifted at the cost of some added notational complexity.

5.1 SPS Conversion

For this analysis we assume a simple programming language with methods or functions:

$$
\begin{aligned}
P &\rightarrow \text{function } f(a_1,\ldots,a_n) = E \\
P &\rightarrow P\,P \\
E &\rightarrow p.g(x_1,\ldots,x_m) \\
E &\rightarrow E + E \\
E &\rightarrow \text{let } v = E \text{ in } E \\
E &\rightarrow \text{BeginPrivilege } E \\
E &\rightarrow \text{CheckPrivilege}(T)
\end{aligned}
$$

where a program is a collection of function definitions; a function body contains function/method calls as well as arithmetic expressions and (not shown here) sequencing statements. The (BeginPrivilege E) statement asserts privileges for the dynamic extent of the execution of E, and CheckPrivilege(T) checks whether the target T is currently accessible. We use lower-case letters to range over program variables and T to stand for target names.

We assume that each method f has an owner (principal) $\mathbf{owner}(f)$. Ownership is associated with code, not classes: in an object-oriented language, if object p belongs to class

$$
\begin{array}{lrcl}
(1) & \mathcal{SPS}_{\text{fun}}(\text{function } f(a_1,\ldots,a_n) = E) & = & \\
& \text{function } f(a_1,\ldots,a_n,s) & = & (\text{let } s' = \mathbf{says}(\mathbf{owner}(f),s) \text{ in } \mathcal{SPS}(E,\ s')) \\
(2) & \mathcal{SPS}(p.g(x_1,\ldots,x_m),\ s') & = & p.g(x_1,\ldots,x_m,s') \\
(3) & \mathcal{SPS}(E_1+E_2,\ s') & = & \mathcal{SPS}(E_1,s') + \mathcal{SPS}(E_2,s') \\
(4) & \mathcal{SPS}(\text{BeginPrivilege } E,\ s') & = & \mathcal{SPS}(E, Ok()) \\
(5) & \mathcal{SPS}(\text{CheckPrivilege}(T),\ s') & = & \mathbf{check}(T,s')
\end{array}
$$

Fig. 2. "Caller-says" SPS conversion.

$$
\begin{array}{lrcl}
(1) & \mathcal{SPS}_{\text{fun}}(\text{function } f(a_1,\ldots,a_n) = E) & = & \text{function } f(a_1,\ldots,a_n,s) = \mathcal{SPS}(E,\ s) \\
(2) & \mathcal{SPS}(p.g(x_1,\ldots,x_m),\ s) & = & p.g(x_1,\ldots,x_m,\mathbf{says}(\mathbf{owner}(p.g),s)) \\
(3) & \mathcal{SPS}(E_1+E_2,\ s) & = & \mathcal{SPS}(E_1,s) + \mathcal{SPS}(E_2,s) \\
(4) & \mathcal{SPS}(\text{BeginPrivilege } E,\ s) & = & \mathcal{SPS}(E, Ok()) \\
(5) & \mathcal{SPS}(\text{CheckPrivilege}(T),\ s) & = & \mathbf{check}(T,s)
\end{array}
$$

Fig. 3. "Callee-says" SPS conversion.

C_2, which inherits method-body f from superclass C_1, then **owner**$(p.f)$ is C_1, not C_2. In principle, this could go either way, but the designers chose to use the concrete implementation's owner because it is easier to ascertain at runtime, and it avoids the danger that a privileged method may call what it thinks is a method of the same class, yet is actually a method in a subclass. This would be an example of a confused deputy problem which we wish to avoid (see section 1).

Figure 2 shows the rules for converting a program to security-passing style. The conversion $\mathcal{SPS}_{\text{fun}}$ is applied to each function; $\mathcal{SPS}$ is applied to each expression. Rule 1 involves the introduction of new local variables s and s' whose names are not used elsewhere. The function f is rewritten to take s as a new formal parameter, the *security context*, which will be the representation of a statement in the ABLP logic. We then construct a new security context with $s' =$ **owner**(f) **says** s, and $\mathcal{SPS}$-convert the body of the function using s' for all outgoing function calls.

Rule 2 of Figure 2 shows the use of s' as the "extra" argument of an outgoing call; rule 3 shows that most statements are unaffected by SPS-conversion. Rule 4 shows that BeginPrivilege discards the security context s' and simply uses *Ok*(); rule 5 shows that CheckPrivilege invokes the decision procedure *check*, described in section 3.5.

To complete the definition of SPS conversion, we assume that the `main` function of the converted program is called with a security context allowing no access to any target (following Netscape) or full access to every target (following Sun).

We have two variants of SPS-conversion; Figure 2 shows a "caller-says" convention, in which a call from g to f involves a computation by g (the caller) of **says**(**owner**$(g),s)$. Figure 3 shows a "callee-says" convention, in which a call from g to f involves a computation by g of **says**(**owner**$(f),s)$.

Either of these conventions is semantically equivalent to stack inspection, but slightly different compile-time optimizations apply, as we will show. For example, in caller-says, **owner**(this.f) can always be computed at compile time; but in callee-says SPS conversion of an O-O language, **owner**(p.g) requires either dynamic method lookup or static analysis.

5.2 Rewriting Java Bytecodes

Stack inspection was originally implemented at Netscape (and then at Sun and Microsoft) by adding support for it to the runtime system. These extensions required changing the stack frame representation, which in turn affected the garbage collector and JIT compiler.

With SPS conversion, we can express the stack-inspection security architecture in "vanilla" Java bytecodes (or source), without stack-frame marks or any other constraints on the Java virtual machine implementation. Every method has an extra parameter for passing the security context, but this parameter and its representation are just Java. SPS conversion is also easier to conceive in languages beyond Java. If the language has function calls, it should be amenable to SPS conversion.

Actually doing the SPS conversion, in practice, turns out to be trickier. Our implementation, with all its warts, is described in section 7.

6. OPTIMIZATION

Netscape and Sun's implementation of stack inspection — by marking frames at BeginPrivilege() and scanning frames in CheckPrivilege() — has very low cost for the vast majority of methods, which do not perform either operation. There is a difficult-to-measure cost of their scheme, in that it may inhibit useful optimizations such as inlining, tail-call optimization, and certain interprocedural optimizations. Also, their system has a linear-time cost for CheckPrivilege(), which must scan a potentially unbounded number of frames to recover the security context. After this, with the current Java 2.0 semantics, analyzing the security context could be potentially as bad as $O(N^2)$ in the number of targets because Sun allows the target speaks-for graph to change over time. If Sun allowed caching of the speaks-for graph (see section 3.6.1), then a transitive closure could be computed once, allowing the final security check to be linear in the size of the security context, which will typically be quite small.

Our semantics costs $O(1)$ per operation (with the same caveats about checking privileges), since a security context has a bounded-size representation. Even so, in order to achieve competitive performance we must minimize the constant-time overhead on each method call. We achieve this by a combination of static optimizations and dynamic caching. By applying static analysis to the full program before it begins running, we can optimize away many of the computations of a new security state.

6.1 Static Optimizations

Suppose a function body $g(\ldots,s)$ contains a call $f(\ldots,s')$, where s and s' are the security-context arguments. The method g must compute $s' = \mathbf{says}(\mathbf{owner}(g),s)$ (in a caller-says convention) or $s' = \mathbf{says}(\mathbf{owner}(f),s)$ (in a callee-says convention). Under certain circumstances, we can let $s' = s$:

—In caller-says, we know that f must compute $s'' = \mathbf{says}(\mathbf{owner}(f),s')$ and then perform operations on s''; f cannot use s' in any other way. If $\mathbf{owner}(g) \Rightarrow \mathbf{owner}(f)$, that is, if the privileges of g are a superset of the privileges of f, then

$$\begin{array}{rcllcl} s'' & = & \mathbf{says}(\mathbf{owner}(f),s') & & \equiv & \mathbf{owner}(f)\ \mathbf{says}\ s' \\ & = & \mathbf{says}(\mathbf{owner}(f),\mathbf{says}(\mathbf{owner}(g),s)) & & \equiv & \mathbf{owner}(f)\ \mathbf{says}\ \mathbf{owner}(g)\ \mathbf{says}\ s \\ & = & \mathbf{says}(\mathbf{owner}(f),s) & & \equiv & \mathbf{owner}(f)\ \mathbf{says}\ s \end{array}$$

by virtue of the fact that $\mathbf{owner}(g) \Rightarrow \mathbf{owner}(f)$, allowing substitutions based on the ABLP axioms. As a result, g can call f with the security context s instead of s', elimi-

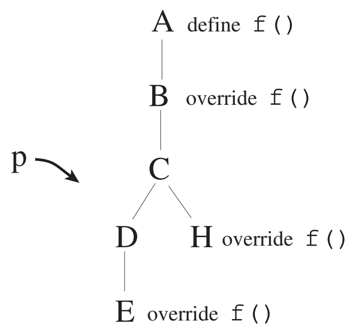

Fig. 4. Class hierarchy analysis.
Variable p of static type C may point to an object of class C, D, E, or H; the owner of $p.f()$ may be the owner of B, E, or H.

nating the computation of the **says** function inside g.

—In callee-says, we know that s was constructed by the caller of g as $s = \textbf{says}(\textbf{owner}(g), t)$. If $\textbf{owner}(g) \Rightarrow \textbf{owner}(f)$, then

$$\begin{array}{rcll} s' & = & \textbf{says}(\textbf{owner}(f), s) & \equiv \quad \textbf{owner}(f) \textbf{ says } s \\ & = & \textbf{says}(\textbf{owner}(f), \textbf{says}(\textbf{owner}(g), t)) & \equiv \quad \textbf{owner}(f) \textbf{ says owner}(g) \textbf{ says } t \\ & = & \textbf{says}(\textbf{owner}(g), t) & \equiv \quad \textbf{owner}(g) \textbf{ says } t \\ & = & s & \end{array}$$

again using the ABLP axioms. As a result, g can call f with the security context s instead of s', eliminating the computation of the **says** function inside g.

In practice, it is very common for one function to call another with the same owner; in such cases, no **says** computation is necessary (since $\textbf{owner}(f) \Rightarrow \textbf{owner}(f)$).

6.1.1 *Criterion for choosing caller-says vs. callee-says.* Caller-says requires calculation of the owner of the currently executing code, and can be statically optimized if the caller is known to speak for the callee. Callee-says requires fetching the owner of the callee, and can be statically optimized if the callee speaks for the caller. Depending on how often these different speaks-for relations can be statically determined, and how often the owner of the callee can be determined statically, one convention or the other may turn out to perform best in practice. We only implemented the callee-says style.

6.1.2 *Static identification of ownership in class-based object-oriented languages.* In an object-oriented program, a program variable `p` declared to be of class `C` may point to an object of any subclass of `C`. Therefore, the method call `p.f()` may invoke any of several actual method bodies, depending upon how `f` is overridden in the subclasses of `C`.

A static analysis of the program may be able to narrow the set of possible types that `p` may take on at the site of the call `p.f()`, and this in turn narrows the set of possible method bodies (callees) that this call site may invoke. Such an analysis can speed up a conventional object-oriented program because dynamic method lookup is more expensive than a static procedure call; if the set of callees can be narrowed to a singleton, then the call `p.f()` can be implemented without run-time lookup. Other kinds of program optimization – interprocedural dataflow analysis, or function-call inlining – also benefit from knowledge of which method-body is called.

For security-passing style, it is not necessary to narrow the set of possible method bodies to a singleton – it suffices to prove that all possible method bodies for this call to `f` have a common owner. In fact, an even weaker property will suffice: for caller-says, we require only that every possible owner of `f` have (nonstrictly) fewer privileges than the owner of its caller, `g`; for callee-says, we require that all owners of `f` have (nonstrictly) more privileges than the owner of `g`.

There has been much work on static analyses of object types. *Class hierarchy analysis* [Fernandez 1995; Dean et al. 1995] simply examines all the subclasses of `C` to see if any of them overrides method `f`. If not, the definition of `f` in class `C` (or, if `C` does not define `f`, the definition of `f` in an ancestor class) must be the callee.

Figure 4 illustrates a simple *flow-insensitive class hierarchy analysis*. Given a variable `p` of static type `C`, we analyze a call `p.f()` as follows. From `C`, we walk up the class hierarchy tree to find the lowest (improper) ancestor of `C` that implements or overrides `f()`, and put that ancestor into the set P. Then we examine all (direct and indirect) subclasses of `C`, and any of those that override `f()` are also put into P.

Information gained from dataflow analysis can prune the set of object types that `p` may contain at the call site; this in turn prunes the set of possible method bodies for `f` at this point, which in turn allows more precise dataflow analysis. This iterative process is called *interprocedural class analysis* and has been shown practical by at least two independent sets of authors [Diwan et al. 1996; DeFouw et al. 1998].

6.1.3 *System speaks for everyone.* Some access-control matrices contain a principal *System* that has access to every target. Even without flow analysis or hierarchy analysis the compiler can use the rule $System \Rightarrow C$ to eliminate **says** computations when System code is calling other methods (in the caller-says convention) or other code is calling System code (in callee-says).

6.1.4 *Leaf procedures.* Many functions do not use their security context in any way. A *leaf procedure* is one that makes no other function call and contains no CheckPrivilege operations; its security context argument is statically dead at all times. A *generalized leaf procedure* is one that neither calls CheckPrivilege nor any native methods, either directly or indirectly. Static analysis of the dynamic call tree can conservatively identify many generalized leaf procedures; these procedures do not require any security-context argument or a **says** computation.

The generalized leaf procedure analysis works by recursively following `invoke` bytecodes from every method (to a limited recursion depth) and is repeated until a fixed point is reached. In practice, thousands of methods can be analyzed this way in less than one CPU second.

This optimization is just a form of interprocedural dead code elimination, and can be done by a conventional object-oriented compiler (after SPS conversion) because security-passing style has expressed all the **says** computations in the underlying programming language. However, the compiler needs to know that **says** has a declarative/functional semantics: calls to **says** can be deleted if the result is dead, even though **says** might have internal side effects to lazily compute part of the transition graph.

Unfortunately, the `invoke` bytecode instructions are not sufficient for the leaf analysis. If a `getstatic` or `putstatic` bytecode references a class that has not yet been loaded, this will cause the class to be *initialized*, creating an implicit call to the target class's initialization method. Likewise, various runtime exceptions, such as indexing beyond an array's boundary or dereferencing a null pointer, will throw exceptions. To throw an exception, an implicit call to the exception's constructor would occur.

These implicit method invocations are not directly visible from Java bytecode, making it difficult to preserve the semantics of the original Java stack inspection system. Our implementation has the class initializers implicitly enable their privileges when they begin running. While this breaks compatibility with Sun's implementation, it still preserves the Java language semantics, which make no guarantees when class initializers are invoked.

The implicit calls to create runtime exceptions are much simpler. By observation, all of the exceptions that might implicitly be thrown make only one native method call, to fill in their stack trace.

In both cases of implicit method invocation, our design breaks with compatibility from Sun's implementation, in favor of allowing more methods to be generalized leaf methods, and thus further reducing the overhead of processing security contexts.

6.2 Dynamic Optimizations

There are a finite number of security contexts s, each corresponding to a subset of the n principals in the system. For a simple browser-applet system, $n = 2$. Each context can be represented with a finite table of labeled out-edges, so that $\mathbf{says}(o,s)$ is computed by looking up o in the table for s.

Although n is bounded, it may not always be tiny (*e.g.,* a stock market with thousands of principals), so we lazily compute the tables and represent only those security contexts that are actually reached. Following an untraversed edge requires (1) looking up a "new" subset in a global hash table to see if this context has been reached before, (2a) using the context-pointer from the table or (2b) creating a new context data structure, and (3) installing the edge into the context that had lacked the edge.

From a security context s there be many consecutive **says** computations by the same principal. In the representation of s we maintain a dynamic one-word cache (o,s') indicating that the most recent **says** calculation on s was $\mathbf{says}(o,s) = s'$. This should speed up the common case.

6.3 Open vs. Closed World Assumptions

Our system, as we have described it, currently makes a fundamental assumption that we can inspect all code before execution begins. This is often called a "closed world assumption." In systems where Java's security features are often used, such as Java applets or servlets, new code may arrive at any time. Currently, all of our algorithms have been designed for a closed world. In particular, our class hierarchy analysis runs once, up front, and code is then generated based on properties true in the closed world. Dean *et al.* [Dean et al. 1995] discusses precisely this issue and proposes a scheme for incrementally updating the analysis.

Keep in mind that the performance numbers in section 7 are based on a closed world. Generally speaking, an open world has strictly less information available from which to infer that an optimization is legal. This implies that, in general, an implementation of security-passing style built for an open world would have strictly worse performance than one built for a closed world, although it may be possible for an open world system to closely approach the performance of a closed world system. For example, in a Java system supporting dynamic code recompilation, such as Sun's HotSpot [Griswold 1998], it would be possible for an SPS incremental analysis to determine that certain classes, previously compiled using optimizations that are now invalid, should now be recompiled. If code from a now-invalid class was inlined in other compiled classes, the recompilation could prove quite expensive. The cost of recompilation would likely dwarf the analysis time (which currently takes less than a CPU second).

7. IMPLEMENTATION AND PERFORMANCE

We have implemented SPS conversion as a transformation on a collection of Java class files. Our implementation was built using the JOIE library (Java Object Instrumentation Environment) from Duke University [Cohen et al. 1998]. This library presents a relatively high-level interface to parse and edit Java class files. We refer to our implementation as SAFKASI – the security architecture formerly known as stack inspection.

We have written approximately 1700 lines of Java code to do static analysis, and 2300 lines to do byte-code rewriting (SPS conversion). Our runtime support (implementing the **says** and CheckPrivilege functions) is 1900 lines. Our system loads, analyzes, and rewrites roughly 800 Java classes in 100 seconds. We made no effort to tune the performance of the rewriter itself; achieving an order of magnitude improvement in rewriting speed should not be unreasonably difficult. JOIE, in particular, uses primitive data structures to represent Java bytecodes. Code insertion in JOIE has $O(N^2)$ cost.

We have implemented flow-insensitive class hierarchy analysis to eliminate **says** computations, and we remove **says** computations from both simple and generalized leaf methods. Because we require the full program for this analysis, we cannot presently support the dynamic loading features of Java (see section 6.1). Instead, we run the program from local disk with our specialized classes.

Our system runs by modifying the class libraries of the NaturalBridge BulletTrain Java compiler [NaturalBridge, LLC 1998]. BulletTrain uses a traditional static compiler to produce native machine executables, in contrast to the dynamic just-in-time compilation used by other Java implementations. BulletTrain currently requires the whole program to be available at compile-time. We chose to use BulletTrain because its authors offered us invaluable assistance with their product. Also, because BulletTrain has an aggressive code optimizer which uses whole-program analysis, we believe performance numbers measured today with BulletTrain will represent what other Java systems will achieve in the future.

7.1 Making SPS Work

Security-passing style has some very nice theoretical properties, but actually implementing it requires a number of difficult cases to be handled properly.

7.1.1 *Native methods.* Java programs can call *native methods* (functions not written in Java) that might then call back to Java methods. We cannot apply SPS conversion within the native methods. Instead, when calling from Java to native, we store the security context s into a per-thread global variable; when calling from native back to Java we fetch s as the security context for the Java code. If we assume that all native method calls have the owner, *System*, then since $\mathbf{says}(System, s) = s$ this is the correct behavior.

We must also support up-calls, where native methods choose to call back into Java methods. The standard mechanism for this, JNI (Java Native Interface), requires the native code to specify a method's complete signature, including the types of its arguments and return value. This means the SPS-converter must generate stubs with the original signatures to receive a JNI up-call. A stub method will retrieve the per-thread stored security context and then invoke its SPS-converted sibling.

7.1.2 *Reflection.* Ideally, the security-passing transformation should not be visible in any way to an application. The Java *reflection* API allows a program to learn how many parameters each of its methods takes; since SPS conversion introduces extra arguments, this is a problem that would have to be fixed by modifying the implementation of reflection; we have not yet done this.

7.1.3 *Bootstrapping.* In practice, bootstrapping proved to be the most difficult aspect of implementing security-passing style. In the BulletTrain system, the majority of the bootstrapping code is written in Java itself. This makes the system extremely sensitive about the order in which classes begin execution, and many classes which appear to be normal are handled specially by the compiler. To address these concerns, an SPS-converted program must bootstrap in three stages.

Classes involved in the very beginning of bootstrapping the runtime were identified by hand and added to a list of classes that are not modified by the SPS converter. Instead, any calls to these classes are treated the same as calls to any native method, storing the security context into a per-thread global variable.

In the second phase of bootstrapping, some SPS-converted classes begin execution but the SPS runtime itself is not yet initialized. Still, SPS-converted classes require a non-null instance of security context be passed to them. To avoid this chicken-and-egg problem, a "dummy" security context, later subclassed to implement the real security context, is created.

Finally, when "real" security contexts are available, the application's main routine can be invoked with a proper security context and execution continues normally.

7.1.4 *Consistency and inheritance.* Because many system classes must not be SPS-converted, an issue arises when an SPS-converted class subclasses a non-converted class or vice versa. It is obviously important to maintain the consistency of the type system, and SPS-converting only a subset of the classes can cause confusion.

To solve the problem, we adopted a rule that, if a class is SPS-converted, then all subclasses of it must also be SPS-converted. Likewise, if an interface is SPS-converted, then all classes that implement that interface must also be SPS-converted.

This rule implies that, if a class *cannot* be SPS-converted, its superclass may not be converted, either. Therefore, several core classes, include `java.lang.Object`, execute unchanged. If a method in an SPS-converted class wishes to call a method in a non-SPS-converted class, it treats the call in the same way native methods are handled: the security context is saved and the method is invoked without the security context argument. In practice, the number of times the security context is saved can be quite significant. See section 7.2.2 for details.

Several issues must still be resolved to make this work. One specific problem was that `java.lang.Thread`, which must not be SPS-converted (it's used very early in the system bootstrapping process), implements the `java.lang.Runnable` interface, which we want to SPS-convert, as it's used throughout the system after bootstrapping. This issue does not occur anywhere else, so it was solved by adding a new method specifically to `java.lang.Thread` during SPS conversion.

Another problem arises when an SPS-converted subclass inherits a method from a non-SPS-converted superclass without overriding it. Normally, the callee in such cases is the

	No Security (baseline)	Stack Inspection	Security-Passing Style
No **says** (leaf)	0	0	1-4 cycles
says$(o,s) = s$ (static opt.)			1-4
says(o,s) (cache hit)			33
says(o,s) (cache miss)			69
BeginPrivilege	24 cycles	2200 cycles	57

Table 1. Measured cost of SPS primitives.

The **says** function is shown as it would be implemented in a leaf method (no security-context argument), in a non-leaf with identical caller and callee owners, and (without static optimization) with and without a hit in the one-word cache.

BeginPrivilege includes the cost of invoking an interface method, as part of the latest Java 2.0 semantics.

Cycle counts were measured by timing microbenchmarks, then dividing by the computer's clock cycle.

superclass's method. However, under SPS conversion, the non-SPS-converted superclass does not have a method with the appropriate SPS-converted signature and neither does the subclass. We address this concern by generating a stub method in the subclass to explicitly delegate to the superclass, when necessary.

7.1.5 *Portability.* As mentioned above, Java systems are relatively fragile during the bootstrapping process. This requires a number of classes to be handled specially. Running SPS-converted code in a different Java environment would require assessing which classes need to be handled specially. Also, sufficient access to the system bootstrapping process is required such that the SPS system can be loaded as early as possible. Aside from these issues, the SPS runtime should be straightforward to add to any Java system.

7.1.6 *Production vs. prototype implementations.* Our prototype implementation makes a number of simplifying assumptions that would not be acceptable in a production system. We make no attempt to support Java's reflection API. We likewise make no attempt to protect the security contexts being passed through a method from the method itself. A carefully-crafted method would be able to manipulate the security context that is inserted within it in our prototype.

Supporting reflection against SPS converted code would be straightforward. It would simply require teaching the reflection calls to ignore any of the added SPS methods or members and to pass along the security context. Protecting the SPS system itself against attack would be trickier. Ideally, we would invoke the Java bytecode verifier first, checking that the class is self-consistent and well-behaved in its original state. After verification, we could then safely perform the SPS conversion.

7.2 Performance

To accurately measure the performance of the original stack inspection primitives as well as their SPS-converted equivalents, we created a series of microbenchmarks that repeatedly enable a privilege, perform a number of recursive method calls, then check the privilege.

All benchmarks were measured on a PC with 384MB of RAM and a 450MHz Intel Pentium II running Windows NT Workstation 4.0 and a development version of the NaturalBridge BulletTrain Java compiler similar to the released version 1.6.

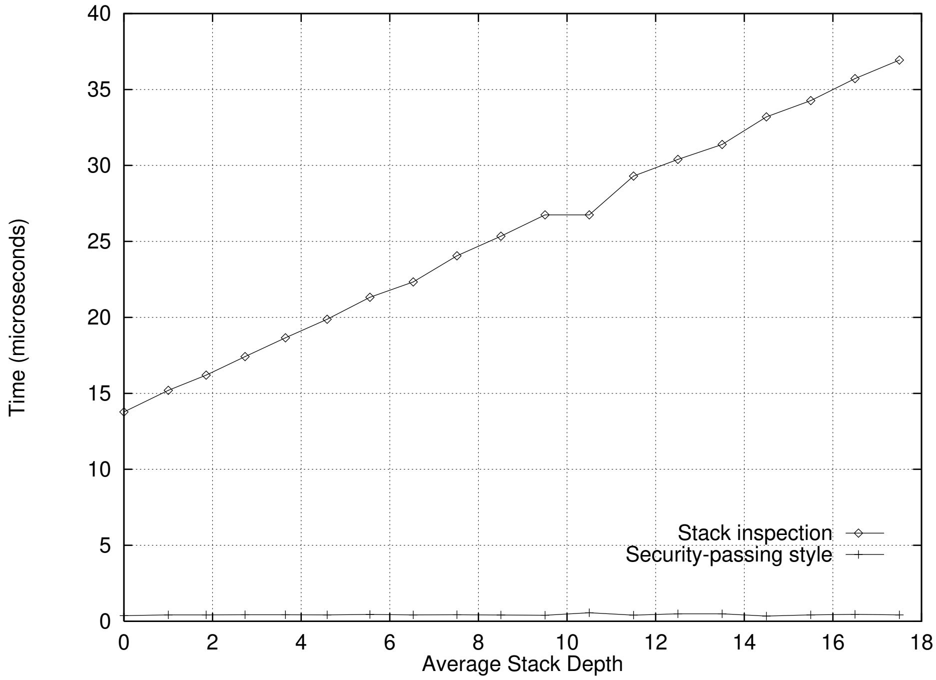

Fig. 5. Cost of CheckPrivilege() microbenchmark.
The times for the SPS check privilege calls are approximately 0.5μsec. This microbenchmark compares NaturalBridge's internal stack inspection system with our security-passing style implementation. The microbenchmark is an implementation of the recursive solution to the Towers of Hanoi problem. On the benchmark machine, 0.5μsec is approximately 225 CPU cycles.

7.2.1 *Micro benchmarks.* We used the measured run times of our microbenchmarks to calculate the cycle count of each stack-inspection primitive in each of three implementations: the null implementation (no security passing, no security checking); the BulletTrain implementation of stack inspection; and our security-passing style. Each microbenchmark was executed ten million times, allowing Java's millisecond-accurate timer to resolve single-cycle differences in execution time. Table 1 shows the results. Figure 5 shows the variable cost of the CheckPrivilege() primitive when using stack inspection compared to the constant-time cost of CheckPrivilege() with security-passing style. The performance difference varies linearly from a factor of 35 to a factor of 88, depending on the stack depth.

7.2.2 *Macro benchmarks.* Despite the success of security-passing style on microbenchmarks, the per-method overhead is more costly when running real applications. Table 2 compares SPS-converted code, with its cheap security checks, to normal code, performing expensive stack inspections for its security checks.

Four benchmark programs were used to test the performance of SPS-converted code to normal code using traditional stack inspection. These benchmarks are SAFKASI, RecRead, Javac, and Jess.

SAFKASI The program which implements the SPS conversion. SAFKASI is benchmarked converting a large Java program (actually, it's converting itself). It is reading in 1172

	No Security (baseline)	Normal Stack Inspection	SPS Conversion (no checks)	SPS Conversion (with checks)
SAFKASI				
Runtime (s)	150.10	261.35	195.14	303.81
Stddev	0.91%	0.85%	0.45%	0.23%
SPS Overhead			30.01%	16.24%
Cost Per Check (ms)		3.95		3.85
RecRead				
Runtime (s)	2.42	10.02	2.43	10.08
Stddev	1.35%	1.13%	1.33%	1.11%
SPS Overhead			0.30%	0.57%
Cost Per Check (ms)		0.74		0.75
Javac				
Runtime (s)	7.74	8.43	9.40	10.02
Stddev	2.00%	2.66%	2.48%	1.73%
SPS Overhead			21.50%	18.86%
Cost Per Check (ms)		0.77		0.69
Jess				
Runtime (s)	3.14	3.18	3.62	3.64
Stddev	2.00%	1.16%	2.60%	2.36%
SPS Overhead			15.31%	14.51%
Cost Per Check (ms)		3.48		1.89

Table 2. Runtime performance of benchmark programs.

Each benchmark program was run in five configurations. The first was instrumented to capture profiling information. The remaining four represented: the normal programming (running unmodified), the normal program with a non-null SecurityManager (and thus performing a stack inspection operation on every security check), and the SPS-converted program with and without security checks.

These benchmarks allow us to measure the overhead of SPS conversion, both without and with security checks being performed. Additionally, since our system counts the number of security checks made, we can compute the average cost per security check, both with stack inspection and with SPS.

	SAFKASI	RecRead	Javac	Jess
Leaf%	10.90%	23.68%	28.55%	37.30%
StaticOpt%	9.89%	76.16%	28.89%	38.00%
DynHit%	79.21%	0.16%	42.56%	24.71%
GetContext%	2.38%	8.46%	5.81%	0.50%
StoreContext%	24.83%	111.62%	15.97%	3.18%

Table 3. Runtime statistics for benchmark programs.

This table represents the effectiveness of static optimization for SPS-converted programs. *Leaf%* represents the number of method calls, at runtime, where the system had statically determined that the callee was a generalized leaf procedure, and therefore needed no additional security context argument. *StaticOpt%* represents the percentage of method calls where the system could statically determine the callee's principal and could thus avoid changing the security context. *DynHit%* represents the percentage of method calls where the system needed to compute a change to the security context but the result had been previously cached. In all these benchmarks, the sum of these percentages is virtually 100%.

Furthermore, we also measured the number of times our system chose to save or restore the security context (*GetContext%* and *StoreContext%*). Normally, this is done before or following a call to a native method or any other method that cannot be rewritten. These measurements are presented as percentages relative to the total number of method calls performed by the benchmark program. So, when RecRead shows a *StoreContext%* of 111.62%, this implies that, on average, each method body made 1.1 calls to store the SPS context prior to invoking a native or otherwise unrewritable method.

`.class` files, analyzing them, and writing out 1172 new `.class` files.

RecRead A simple recursive program that, given a directory as input, recursively iterates through all the subdirectories and reads the first 1024 bytes of every file. This program should be I/O bound, and should also generate a large number of security checks relative to the amount of CPU is consumes. RecRead was always run with a warm file cache.

Javac The compiler from Sun's Java 2.0 distribution. For this benchmark, Javac is compiling Jess from scratch (all `.class` files are deleted between benchmark runs).

Jess The Java Expert System Shell, version 5.0 [Friedman-Hill 1997]. For this benchmark, we asked jess to evaluate `hard.clp`, which is included as an example in the Jess distribution.

Each benchmark configuration was executed ten times and average results are presented here. We show performance numbers with the average and standard deviation of their runtimes.

On these benchmarks, we found the cost of performing a security check to be roughly the same, whether using SPS-converted code or traditional stack inspection (with a small edge given SPS-converted code). However, the general performance overhead of introducing SPS-conversion to programs making no security checks varies from 15-30%. The cheaper security checks with SPS only reduce this to a range of 15-19%.

Interestingly, the cost per security check, as measured on the benchmarks, was not constant for SPS-converted code. The costs per check for SPS and for stack inspection tended to mirror one other. If one was cheap, the other would be cheap. In fact, the cost of a security check was orders of magnitude higher than the cost of either primitive stack inspection or the SPS check alone. This implies that other machinery in the SecurityManager is the performance bottleneck for security checks. The performance overhead of the SecurityManager is particularly apparent in the RecRead benchmark, where the security checks caused the benchmark to run below one quarter of its original speed.

Furthermore, both our security-passing system and the BulletTrain stack inspection system are prototype implementations. Both "cheat" by returning `ProtectionDomain` structures which grant permission for any request and neither has been heavily tuned for performance. Our benchmarks imply that, while SPS conversion can slow down a program by up to 30%, the other costs associated with making security checks can dominate the system's performance for programs performing significant security checks.

The culprit for the slowdown when enabling security checks is the Java 2.0 permission checking system [Gong 1999]. Java 2.0 has defined an extensible system where one permission can imply another, requiring a significant amount of bookkeeping to track all the known permissions in the system. For programs, such as RecRead, where these costs dominate system performance, the performance difference caused by SPS conversion is largely irrelevant. If the permission checking system were heavily tuned or redesigned, then the performance difference between SPS and stack inspection would be more relevant. In particular, if programs like Jess, where SPS security checks have half the cost of stack inspection, were to perform more security checks, as done in RecRead, SPS could conceivably result in a faster system than stack inspection. In the current system, as it stands, the high runtime overhead of SPS-conversion is never paid for by the lower per-security-check costs.

7.2.3 *Optimization effectiveness.* Section 6 describes a series of static and dynamic optimizations that can be performed as part of the SPS-conversion process. Table 3 describes how well these optimizations worked in practice. The generalized leaf procedure analysis combined with static determination of a method callee's principal allowed anywhere from 20-99.84% of all method call SPS conversions to be optimized away. In the worst case, our dynamic optimizations rarely missed. We can conclude that our optimizations were extremely effective.

The other result that appears in our analysis is the prevalence of the need to store and retrieve the security context. When a callee cannot be SPS-converted (see the discussion above on bootstrapping and native methods), the security context must be saved. Likewise, if control flow returns from one of these methods to a normal SPS-converted method, the security context must be restored. Expressed as a percentage of the total number of method calls performed, our benchmarks show a significant number of StoreContext calls (ranging from 3-24% in normal circumstances to over 100% in a degenerate case that performs only I/O operations) and dramatically fewer GetContext calls (ranging from 0.5-8.5%). This implies that the majority of times in which we store the context, it is never retrieved. Our system could be further optimized by identifying, by hand, which of the non-SPS-converted methods never perform up-calls to SPS-converted classes. With profiling enabled, our system counts the number of times a security context is either saved or restored as a function of the caller. This list can be used to identify hot spots for further analysis.

8. RELATED WORK

Security mechanisms can be generally broken down into those concerning access controls (access to specific resources such as files and network connections), resource limits (CPU and memory use, network bandwidth, and other resources where individual use is cheap but aggregate use over time is costly), and information flow.

Language-based mechanisms have been used for security purposes before, perhaps as early as the Burroughs B5000 series computers [Burroughs Corporation 1969]. Many more recent systems, including Smalltalk [Goldberg and Robson 1989], Pilot [Redell et al. 1980], Cedar [Swinehart et al. 1986], Lisp Machines [Bromley 1986], and Oberon [Wirth and Gutknecht 1992] have taken advantage of language-based mechanisms to provide access control services. More recent systems, such as MzScheme [Flatt et al. 1999] and J-Kernel [Hawblitzel et al. 1998; Spoonhower et al. 1998] use name-space management or capability-style semantics for access control.

A number of projects have addressed the issue of resource management within language runtimes. Back, Tullmann, Stoller, Hsieh, and Lepreau [2000], Back and Hsieh [1999], and van Doorn [2000] discuss developing an operating system-like environment within Java. They discuss implementing virtual kernel-user boundary within the Java API and the JVM. Likewise, JRes [Czajkowski and von Eicken 1998] and Bernadat, Lambright, and Travostino [1998] supports controls on memory usage.

Some systems, such as PLAN [Hicks et al. 1998], restrict the language to guarantee that programs will terminate or otherwise behave in some fashion. Similar restrictions can also be enforced with proof-carrying code [Necula and Lee 1996; Necula and Lee 1997].

JFlow [Meyers 1999] enforces information flow policies against Java programs.

The system presented here, SAFKASI, is built using Java bytecode rewriting. Bytecode rewriting is a very general technique that has been used in a number of systems. J-Kernel, JRes, Naccio [Evans and Twyman 1999], and Kimera [Sirer et al. 1999] all use Java bytecode rewriting as part of their security architecture.

9. CONCLUSIONS

Stack inspection has proven to be a useful technique for managing the security requirements of mobile code systems such as Java and has been adopted by all the major Java software vendors. We present a formal model of stack inspection based on a belief logic. We present the design and implementation of security-passing style, and a proof of its equivalence to stack inspection. We analyze the performance of our implementation. We discuss a number of optimizations for security-passing style, both static optimizations based on a class hierarchy analysis, and dynamic optimizations based on caching of previous results.

While the performance of our prototype is not as good as the original stack inspection system, we now have a powerful and theoretically sound system with equivalent semantics to stack inspection, yet which may be more easily adapted to other languages beyond Java. Security-passing style addresses concerns that mobile code systems must require ad-hoc changes to language runtimes or must rely on traditional operating systems mechanisms. As mobile code is increasingly deployed, whether in the form of active networks, shared virtual realities, or programmed stock trading, the importance of a sound security architecture, such as stack inspection, increases likewise.

ACKNOWLEDGMENTS

This work is supported in part by grants from the National Science Foundation (CCR-9457813 and CCR-9985332) and the Alfred P. Sloan Foundation as well as donations from Sun Microsystems, Intel, Microsoft, Bellcore, and Merrill Lynch. We sincerely thank Kenneth Zadeck, David Chase, and Roger Hoover of NaturalBridge L.L.C. for their invaluable assistance in debugging our SAFKASI port to their Java system. We also thank Martín Abadi, Dirk Balfanz, Drew Dean, and the anonymous referees who have helped shape this paper into its current form.

REFERENCES

ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. D. 1993. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems 15*, 4 (Sept.), 706–734.

BACK, G. AND HSIEH, W. 1999. Drawing the Red Line in Java. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in O perating Systems* (Rio Rico, Arizona, March 1999). `http://www.cs.utah.edu/flux/papers/redline-hotos7.ps`.

BACK, G., TULLMANN, P., STOLLER, L., HSIEH, W. C., AND LEPREAU, J. 2000. Techniques for the design of Java Operating System. In *Proceedings of the 2000 Usenix Annual Technical Conference* (San Diego, California, June 2000). `http://www.cs.utah.edu/flux/papers/javaos-usenix00-base.html`.

BERNADAT, P., LAMBRIGHT, D., AND TRAVOSTINO, F. 1998. Towards a resource-safe Java for service guarantees in uncooperative environments. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain, Dec. 1998).

BROMLEY, H. 1986. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers.

Burroughs Corporation. 1969. *Burroughs B6500 Information Processing Systems Reference Manual*. Detroit, Michigan: Burroughs Corporation.

BURROWS, M., ABADI, M., AND NEEDHAM, R. M. 1990. A logic of authentication. *ACM Transactions on Computer Systems 8*, 1 (Feb.), 18–36.

COHEN, G., CHASE, J., AND KAMINSKY, D. 1998. Automatic program transformation with JOIE. In *Proceedings of the 1998 Usenix Annual Technical Symposium* (New Orleans, Louisiana, June 1998), pp. 167–178.

CZAJKOWSKI, G. AND VON EICKEN, T. 1998. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 1998), pp. 21–35.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)* (Århus, Denmark, Aug. 1995).

DEFOUW, G., GROVE, D., AND CHAMBERS, C. 1998. Fast interprocedural class analysis. In *25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1998), pp. 222–236. ACM Press.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically typed object-oriented programs. In *OOPSLA '96: 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Volume 31 (1996), pp. 292–305. ACM Press.

EVANS, D. AND TWYMAN, A. 1999. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (Oakland, California, May 1999), pp. 32–45.

FERNANDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of ACM SIGPLAN '95 Conference on Programming Langauge Design and Implementation* (1995), pp. 103–115.

FLATT, M., FINDLER, R. B., KRISHNAMURTHY, S., AND FELLEISEN, M. 1999. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP '99)* (Paris, France, Sept. 1999). `http://www.cs.rice.edu/CS/PLT/Publications/icfp99-ffkf.ps.gz`.

FRIEDMAN-HILL, E. J. 1997. *Jess, The Java Expert System Shell*. Distributed Computing Systems, Sandia National Laboratories. `http://herzberg1.ca.sandia.gov/jess/`.

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk 80: The Language*. Addison-Wesley, Reading, Massachusetts.

GONG, L. 1999. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts.

GONG, L. AND SCHEMERS, R. 1998. Implementing protection domains in the Java Development Kit 1.2. In *The Internet Society Symposium on Network and Distributed System Security* (San Diego, California, March 1998). Internet Society.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts.

GRISWOLD, D. 1998. *The Java HotSpot Virtual Machine Architecture*. Palo Alto, California: Sun Microsystems. `http://java.sun.com/products/hotspot/whitepaper.html`.

HARDY, N. 1988. The confused deputy. *ACM Operating Systems Review 22*, 4 (Oct.), 36–38. `http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html`.

HAWBLITZEL, C., CHANG, C.-C., CZAJKOWSKI, G., HU, D., AND VON EICKEN, T. 1998. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference* (New Orleans, Louisiana, June 1998). USENIX.

HENNESSY, J. L. 1982. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems 4*, 4 (July), 323–344.

HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. 1998. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages* (1998), pp. 86–93. ACM. `http://www.cis.upenn.edu/~switchware/papers/plan.ps`.

LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. 1992. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems 10*, 4 (Nov.), 265–310.

LAMPSON, B. W. 1971. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems* (Princeton University, March 1971), pp. 437–443. Reprinted in *Operating Systems Review*, 8(1):18–24, January 1974.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts.

MEYERS, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)* (San Antonio, Texas, Jan. 1999), pp. 228–241.

Microsoft Corporation. 1997. *Microsoft Security Management Architecture White Paper*. Redmond, Washington: Microsoft Corporation. `http://www.microsoft.com/ie/security/ie4security.htm`.

NaturalBridge, LLC. 1998. *BulletTrain Java Compiler*. NaturalBridge, LLC. `http://www.naturalbridge.com`.

NECULA, G. C. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)* (Seattle, Washington, Oct. 1996), pp. 229–243.

NECULA, G. C. AND LEE, P. 1997. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, Number 1419 in Lecture Notes in Computer Science (Oct. 1997). Springer-Verlag.

Netscape Communications Corporation. 1997. *Introduction to the Capabilities Classes*. Mountain View, California: Netscape Communications Corporation. `http://developer.netscape.com/library/documentation/signedobj/capabilities/index.html`.

REDELL, D., DALAL, Y., HORSLEY, T., LAUER, H., LYNCH, W., MCJONES, P., MURRAY, H., AND PURCELL, S. 1980. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2 (Feb.), 81–92.

ROSKIND, J. 1996. *Evolving the Security Model For Java From Navigator 2.x to Navigator 3.x*. Mountain View, California: Netscape Communications Corporation. `http://developer.netscape.com/library/technote/security/sectn1.html`.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sept.), 1278–1308.

SAULPAUGH, T., CLEMENTS, T., AND MIRHO, C. A. 1999. *Inside the JavaOS Operating System*. Addison-Wesley, Reading, Massachusetts.

SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. 1999. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles* (Kiawah Island Resort, South Carolina, Dec. 1999), pp. 202–216. ACM.

SPOONHOWER, D., CZAJKOWSKI, G., HAWBLITZEL, C., CHANG, C.-C., HU, D., AND ON EICKEN, T. 1998. Design and evaluation of an extensible web & telephony server based on the J-Kernel. Technical Report 98-1715 (Nov.), Cornell University. `http://www2.cs.cornell.edu/slk/papers/tr98-1715.pdf`.

STEELE, G. L. 1978. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, Massachusetts.

SWINEHART, D. C., ZELLWEGER, P. T., BEACH, R. J., AND HAGMANN, R. B. 1986. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems 8*, 4 (Oct.), 419–490.

TOLMACH, A. P. AND APPEL, A. W. 1990. Debugging Standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (New York, June 1990), pp. 1–12. ACM Press.

VAN DOORN, L. 2000. A secure Java virtual machine. In *Ninth USENIX Security Symposium Proceedings* (Denver, Colorado, Aug. 2000).

WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. 1997. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles* (Saint-Malo, France, Oct. 1997), pp. 116–128.

WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, California, May 1998), pp. 52–63.

WILLE, C. 2000. *Presenting C#*. Sams Publishing, USA.

WIRTH, N. AND GUTKNECHT, J. 1992. *Project Oberon*. ACM Press.

WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. 1994. Authentication in the Taos operating system. *ACM Transactions on Computer Systems 12*, 1 (Feb.), 3–32.

APPENDIX

A. ACCESS CONTROL LOGIC DETAILS

In order to model stack inspection, we chose to adopt a subset of the belief logic from Abadi, Burrows, Lampson, and Plotkin [1993] (hereafter, ABLP logic).

A.1 ABLP Grammar

This section presents a grammar for valid ABLP expressions. Note that, while this grammar is not completely unambiguous, the axioms that operate on ABLP expressions are unambiguous.

There are two fundamental types in ABLP: statements and principals. A statement could be an atomic statement such as "the sky is blue." It could also be a compound statement such as "Bob says the sky is blue" or "Alice speaks for Bob," as indicated with the $\Rightarrow$ symbol. A statement may also be the conjunction of several independent statements, as indicated with the $\wedge$ symbol. Or, a statement may imply another statement, as indicated with the $\supset$ symbol.

$$
\begin{array}{lcl}
\text{Statement} & \rightarrow & \textit{AtomicStatement} \\
\text{Statement} & \rightarrow & \text{Statement} \wedge \text{Statement} \\
\text{Statement} & \rightarrow & \text{Statement} \supset \text{Statement} \\
\text{Statement} & \rightarrow & \text{Principal } \textbf{says} \text{ Statement} \\
\text{Statement} & \rightarrow & \text{Principal} \Rightarrow \text{Principal}
\end{array}
$$

A principal can be an atomic principal such as "Alice" or "Bob." It may also be a compound principal such as "Alice quoting Bob," as indicated with the $|$ symbol or a conjunction of principals, as indicated with the $\wedge$ symbol.

$$
\begin{array}{lcl}
\text{Principal} & \rightarrow & \textit{AtomicPrincipal} \\
\text{Principal} & \rightarrow & \text{Principal} \mid \text{Principal} \\
\text{Principal} & \rightarrow & \text{Principal} \wedge \text{Principal}
\end{array}
$$

To avoid ambiguous statements or at least make statements more legible, parentheses are also acceptable in all the obvious places.

Principal → (Principal)
Statement → (Statement)

Thus, it's perfectly reasonable to make a statement such as

$$((Alice \wedge Bob) \textbf{ says } (Charlie \Rightarrow (Alice \wedge Bob))) \wedge$$
$$(Charlie|Alice \textbf{ says } X) \wedge$$
$$((Alice \textbf{ says } X) \supset X)$$

where the first part is a form of delegation (Alice and Bob delegating their privileges to Charlie), the second part is an assertion that "Charlie quoting Alice" wants to do X (*i.e.,* Charlie is claiming that Alice wants to do X), and the third part is an access control rule stating that when Alice says she wants to do X, we will believe her.

A.2 Axioms

Here is a list of the subset of axioms in ABLP logic used in this paper. We omit axioms for delegation, roles, and exceptions because they are not necessary to discuss stack inspection.

A.2.1 *Axioms About Statements.*

If s is an instance of a theorem of propositional logic then s is true in ABLP. (9)

$$\text{If } s \text{ and } s \supset s' \text{ then } s'. \tag{10}$$

$$(A \textbf{ says } s \wedge A \textbf{ says } (s \supset s')) \supset A \textbf{ says } s'. \tag{11}$$

$$\text{If } s \text{ then } A \textbf{ says } s \text{ for every principal } A. \tag{12}$$

A.2.2 *Axioms About Principals.*

$$(A \wedge B) \textbf{ says } s \equiv (A \textbf{ says } s) \wedge (B \textbf{ says } s) \tag{13}$$

$$(A \mid B) \textbf{ says } s \equiv A \textbf{ says } B \textbf{ says } s \tag{14}$$

$$(A = B) \supset (A \textbf{ says } s \equiv B \textbf{ says } s) \tag{15}$$

$$(A \mid (B \mid C)) \equiv ((A \mid B) \mid C) \tag{16}$$

$$(A \mid (B \wedge C)) \equiv (A \mid B) \wedge (A \mid C) \tag{17}$$

$$(A \Rightarrow B) \equiv (A = A \wedge B) \tag{18}$$

$$(A \textbf{ says } (B \Rightarrow A)) \supset (B \Rightarrow A) \tag{19}$$

So, given the the following statement:

$$((Alice \wedge Bob) \textbf{ says } Charlie \Rightarrow (Alice \wedge Bob)) \wedge$$
$$(Charlie|Alice \textbf{ says } X) \wedge$$
$$((Alice \textbf{ says } X) \supset X)$$

we might try to prove X.

$Charlie \Rightarrow (Alice \wedge Bob)$	by axiom 19
$(Charlie \wedge Alice \wedge Bob)|Alice \textbf{ says } X$	by axiom 18
$(Charlie|Alice \textbf{ says } X) \wedge$ $(Alice|Alice \textbf{ says } X) \wedge (Bob|Alice \textbf{ says } X)$	by axiom 13
$Alice \mid Alice \textbf{ says } X$	by axiom 9
$Alice \textbf{ says } Alice \textbf{ says } X$	by axiom 14
$Alice \textbf{ says } X$	by axiom 11
X	by axiom 11 □

Now, in general, not all ABLP proofs are this easy. It is possible to encode problems in ABLP that are equivalent to the halting problem. However, by carefully choosing a subset of ABLP, we can not only guarantee that proofs are decidable, but we can also make efficient decision procedures for them. Section 3 presents the subset of ABLP that we use to model Java's stack inspection.

A.3 Applying ABLP

With an understanding of how ABLP logic works, we can explain how it can be used to model actual systems. A great amount of detail on this is available in Lampson, Abadi, Burrows, and Wobber [1992]. ABLP can be used to model the flow of control through a single system, from user to keyboard to motherboard to device driver to operating system to user process. It can also be used to model information passing across a network to the same level of detail. The key is quoting. When an application receives a keystroke, it might want to verify that the keystroke, in fact, came from the user. In the model, such an application would be required to validate

$$(\mathit{Kernel}|\mathit{DeviceDriver}|\mathit{Keyboard}\ \textbf{says}\ \mathit{KeyPressed(`g')}) \supset \mathit{KeyPressed(`g')}$$

In order to do this, it must believe that each layer truthfully speaks for the layer below it:

$$\begin{array}{l} \mathit{Kernel} \Rightarrow \mathit{DeviceDriver} \\ \mathit{DeviceDriver} \Rightarrow \mathit{Keyboard} \\ (\mathit{Keyboard}\ \textbf{says}\ \mathit{KeyPressed}(\mathrm{x})) \supset \mathit{KeyPressed}(\mathrm{x}) \end{array}$$

Given the above beliefs and the axioms of ABLP logic, an application may safely believe in the authenticity of its keystrokes.

If we wish to add a network window server (such as X) to this model, we must prove that the window server speaks for the keyboard. Such a proof would require modeling the event dispatch mechanism inside the server. If the window server supported features like synthetic key events (where an application may simulate keystroke events to drive another application), this would also need to be taken into account in the model. As the model's complexity grows, our certainty of keystroke authenticity is dependent on our ability to make proofs as above.

ABLP can be applied to all kinds of authentication problems. A related logic, BAN logic [Burrows et al. 1990], has been applied to the underlying cryptographic protocols as well.