Understanding Java Stack Inspection*

Dan S. Wallach
dwallach@cs.princeton.edu

Edward W. Felten
felten@cs.princeton.edu

Secure Internet Programming Laboratory
Department of Computer Science
Princeton University

Abstract

Current implementations of Java make security decisions by searching the runtime call stack. These systems have attractive security properties, but they have been criticized as being dependent on specific artifacts of the Java implementation.

This paper models the stack inspection algorithm in terms of a well-understood logic for access control and demonstrates how stack inspection is a useful tool for expressing and managing complex trust relationships. We show that an access control decision based on stack inspection corresponds to the construction of a proof in the logic, and we present an efficient decision procedure for generating these proofs.

By examining the decision procedure, we demonstrate that many statements in the logic are equivalent and can thus be expressed in a simpler form. We show that there are a finite number of such statements, allowing us to represent the security state of the system as a pushdown automaton. We also show that this automaton may be embedded in Java by rewriting all Java classes to pass an additional argument when a procedure is invoked. We call this *security-passing style* and describe its benefits over previous stack inspection systems. Finally, we show how the logic allows us to describe a straightforward design for extending stack inspection across remote procedure calls.

1 Introduction

The Java language [7] and virtual machine [11] are now being used in a wide variety of applications: Web browsers and servers, multi-user chat systems (MUDs), agent systems, commerce applications, smart cards, and more. Some systems use Java simply as a better programming language, using Java's type-safety to prevent a host of bugs endemic to C programming. In other systems, Java is also being relied upon for access control. Java's promise, from its initial debut in the HotJava Web browser, has been to allow mutually untrusting code modules to co-exist in the same virtual machine in a secure and controllable manner. While there have been several security problems along the way [4, 13], the security of Java implementations is improving and Java has continued to grow in popularity.

To implement a Java application that runs untrusted code within itself (such as the HotJava Web browser), the Java system libraries need a way to distinguish between calls originating from untrusted code, which should be restricted, and calls originating from the application itself, which should be allowed to proceed (subject to any access controls applied by the underlying operating system). To solve this problem, the Java runtime system exports an interface to allow security-checking code to examine the runtime stack for frames executing untrusted code, and allows security decisions to be made at runtime based on the state of the stack.

While a number of other techniques may be used to achieve the same goals as stack inspection [21], stack inspection has proven to be quite attractive and has been adopted by all the major Java vendors [15, 6, 14] to meet their need to provide more flexible security policies than the rigid "sandbox" policy, which restricted all non-local code to the same set of privileges. Stack inspection is also a useful technique to allow highly-trusted code to operate with less than its full privileges, which can help prevent common program bugs from becoming security holes.

Stack inspection has been criticized for its implementation-specific and seemingly ad-hoc definition, which restricts the flexibility of an optimizing compiler and hinders its applicability to other languages. To address these concerns, we will present a model of

stack inspection using a belief logic designed by Abadi, Burrows, Lampson, and Plotkin [1] (hereafter, ABLP logic) to reason about access control. Using this logic, we will derive an alternate technique for implementing stack inspection which is applicable to Java and other languages. Our procedure applies to remote procedure calls as well as local ones.

This paper is organized as follows. Section 2 begins by reviewing Java's stack inspection model. Next, Section 3 explains the subset of ABLP logic we use. Section 4 shows the mapping from stack inspection to ABLP logic and discusses their equivalence. Section 5 presents a high-performance and portable procedure to implement stack inspection. Finally, Section 6 considers remote procedure calls and shows how stack inspection helps to address remote procedure call security. The appendices list the axioms of ABLP logic used in this paper, and present proofs of our theorems.

2 Java Stack Inspection

This section describes Java's current stack inspection mechanism[1]. Variations on this approach are taken by Netscape's Communicator 4.0 [15], Microsoft's Internet Explorer 4.0 [14], and Sun's Java Development Kit 1.2 [6].

Stack inspection has a number of useful security properties [21] but very little prior art. In some ways, it resembles dynamic variables (where free variables are resolved from the caller's environment rather than from the environment in which the function is defined), as used in early versions of LISP [12]. In other ways, it resembles the notion of *effective user ID* in Unix, where the current ID is either inherited from the calling process or set to the executable's owner through an explicit *setuid* bit.

2.1 Type Safety and Encapsulation

Java's security depends fundamentally on the *type safety* of the Java language. Type safety guarantees that a program may not treat pointers as integers and vice versa and likewise may not exceed the allocated size of an array. This prevents arbitrary access to memory and makes it possible for a software module to encapsulate its state: to declare that some of its variables and procedures may not be accessed by code outside itself. By allowing access only through a few carefully written entry points, a module can apply access control checks to all attempts to access its state.

For example, the Java virtual machine protects access to operating system calls in this way. Only the virtual machine may directly make a system call, and other code must call into the virtual machine through explicit entry points which implement security checks.

2.2 Simplified Stack Inspection

To explain how stack inspection works, we will first consider a simplified model of stack inspection. In this model, the only principals are "system" and "untrusted". Likewise, the only privilege available is "full." This model resembles the stack inspection system used internally in Netscape Navigator 3.0 [17].

In this model, every stack frame is labeled with a principal ("system" if the frame is executing code that is part of the virtual machine or its built-in libraries, and "untrusted" otherwise), and contains a privilege flag which may be set by a system class which chooses to "enable its privileges," explicitly stating that it wants to do something dangerous. An untrusted class cannot set its privilege flag. When a stack frame exits, its privilege flag (if any) automatically disappears.

All procedures about to perform a dangerous operation such as accessing the file system or network first apply a stack inspection algorithm to decide whether access is allowed. The stack inspection algorithm searches the frames on the caller's stack in sequence, from newest to oldest. The search terminates, allowing access, upon finding a stack frame with a privilege flag. The search also terminates, forbidding access and throwing an exception, upon finding an untrusted stack frame (which could never have gotten a privilege flag).

2.3 Stack Inspection

The stack inspection algorithm used in current Java systems can be thought of as a generalization of the simple stack inspection model described above. Rather than having only "system" and "untrusted" principals, many principals may exist. Likewise, rather than having only "full" privileges, a number of more specific privileges are defined, so different principals may have different degrees of access to the system.

Four fundamental primitives are necessary to use stack inspection:[2]

- `enablePrivilege()`
- `disablePrivilege()`
- `checkPrivilege()`
- `revertPrivilege()`

[1] This approach is sometimes incorrectly referred to as "capability-based security" in vendor literature.

[2] Each Java vendor has different syntax for these primitives. This paper follows the Netscape syntax.

When a dangerous resource *R* (such as the file system) needs to be protected, the system must be sure to call `checkPrivilege(`*R*`)` before accessing *R*.

When code wishes to *use* *R*, it must first call `enablePrivilege(`*R*`)`. This consults the local policy to see whether the principal of the caller is permitted to use *R*. If it is permitted, an *enabled-privilege(R)* annotation is made on the current stack frame. The code may then use *R* normally. Afterward, the code may call `revertPrivilege(`*R*`)` or `disablePrivilege(`*R*`)` to discard the annotation or it may simply return, causing the annotation to be discarded along with the stack frame. `disablePrivilege()` creates a stack annotation that can hide an earlier enabled privilege, whereas `revertPrivilege()` simply removes annotations from the current frame.

The generalized `checkPrivilege()` algorithm, used by all three implementations, is shown in figure 1. The algorithm searches the frames on the caller's stack in sequence, from newest to oldest. The search terminates, allowing access, upon finding a stack frame that has an appropriate *enabled-privilege* annotation. The search also terminates, forbidding access (and throwing an exception), upon finding a stack frame that is either forbidden by the local policy from accessing the target or that has explicitly disabled its privileges.

We note that each vendor takes different actions when the search reaches the end of the stack uneventfully: Netscape denies permission, while both Sun and Microsoft allow it.

```
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 (stackFrame has disabled privilege for target)
            throw ForbiddenException;
    }

    // if we reached here, we fell off the end of the stack
    if (Netscape 4.0)
        throw ForbiddenException;
    if (Microsoft IE 4.0 || Sun JDK 1.2)
        return; // allow access
}
```

Figure 1: Java's stack inspection algorithm.

3 Access Control Logic

We will model the behavior of Java stack inspection using ABLP logic [1, 9]. ABLP logic allows us to reason about what we believe to be true given the state of the system and a set of axioms. It has been used to describe authentication and authorization in distributed systems such as Taos [22] and appears to be a good match for describing access control within Java. We use a subset of the full ABLP logic, which we will describe here. Readers who want a full description and a more formal development of the logic should see [1] or [9].

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 a false 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.

  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.

Appendix A gives a full list of the axioms of the logic. This is a subset of the ABLP logic: we omit some of the operators defined by ABLP since we do not need them.

4 Mapping Java to ABLP

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

4.1 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[3] 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 18, this implies that

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

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

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 17) 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. To simplify the discussion, all of our examples will use single signers, but the theory supports multiple signers without any extra difficulty.

4.2 Targets

Recall that 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

$$\forall T \in Targets,\ (T \textbf{ says } Ok(T)) \supset Ok(T) \tag{6}$$

says that T can allow access to itself.

Many targets are defined in relation to services 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.

4.3 Setting Policy

We use a standard access matrix [10] 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. Informally, 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*.

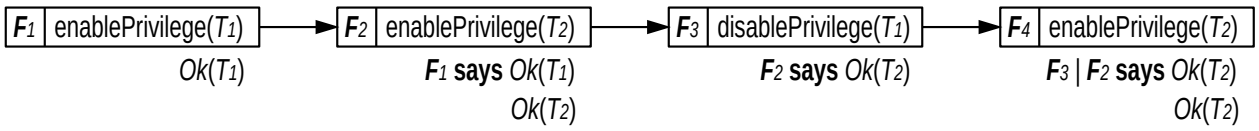

Figure 2: Example of interaction between stack frames. Each rectangle represents a stack frame. Each stack frame is labeled with its name. In this example, each stack frame makes one `enablePrivilege()` or `disablePrivilege()` call, which is also written inside the rectangle. Below each frame is written its belief set after its call to `enablePrivilege()` or `disablePrivilege()`.

4.4 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 *belief set* of F and write it $\mathcal{B}_F$. We now describe where the beliefs come from.

4.4.1 Starting a Program

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

4.4.2 Enabling Privileges

If a stack frame F calls `enablePrivilege(`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 $\mathcal{B}_F$.

4.4.3 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 all of F's beliefs. When F tells G a statement S, the statement F **says** S is added to $\mathcal{B}_G$.

4.4.4 Disabling and Reverting Privileges

A stack frame can also choose to disable some of its privileges. The call `disablePrivilege(`T`)` asks to disable any privilege to access the target T. This is implemented by giving the frame a new belief set which consists of the old belief set with all statements in which anyone says $Ok(T)$ removed. `revertPrivilege()` is handled in a similar manner, by giving the frame a new belief set that is equal to the belief set it originally had. While our treatment of `disablePrivilege()` and `revertPrivilege()` is a bit inelegant, it seems to be the best we can do.

4.4.5 Example

Figure 2 shows an example of these rules in action. In the beginning, $\mathcal{B}_{F_1} = \{\}$. F_1 then calls `enablePrivilege(`T_1`)`, which adds the statement $Ok(T_1)$ to $\mathcal{B}_{F_1}$.

When F_2 is created, F_1 tells it $Ok(T_1)$, so $\mathcal{B}_{F_2}$ is initially $\{F_1$ **says** $Ok(T_1)\}$. F_2 then calls `enablePrivilege(`T_2`)`, which adds $Ok(T_2)$ to $\mathcal{B}_{F_2}$.

$\mathcal{B}_{F_3}$ initially contains $F_2 \mid F_1$ **says** $Ok(T_1)$ and F_2 **says** $Ok(T_2)$. When F_3 calls `disablePrivilege(`T_2`)`, the latter belief is deleted from $\mathcal{B}_{F_3}$. $\mathcal{B}_{F_4}$ initially contains $F_3 \mid F_2$ **says** $Ok(T_1)$. When F_4 calls `enablePrivilege(`T_2`)`, this adds $Ok(T_2)$ to $\mathcal{B}_{F_4}$.

4.5 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, $\mathcal{A}_{VM}$, and $\mathcal{B}_F$ (the belief set of the frame which called `checkPrivilege()`).

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

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

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, there is an efficient decision procedure that gives the correct answer for our subset of the logic. `checkPrivilege()` implements that decision procedure.

The decision procedure used by `checkPrivilege()` takes, as arguments, an environment $\mathcal{E}_F$ and a target T. The decision procedure 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 decision procedure always terminates.*

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

Proofs of these two theorems appear in Appendix B.

Conjecture 1 (Completeness) *If the 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 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 decision procedure described above is equivalent to the Java stack inspection algorithm of section 2.*

A proof of this theorem appears in Appendix B.

4.6 Other Differences

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.

4.6.1 Extension: 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" which 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" which 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 implies a particular other target. This can be handled with a simple extension to the model.

4.6.2 Extension: 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 an empty belief set. In Sun and Microsoft's systems, the first frame on the stack of the new thread is told the belief set of the stack frame that created the thread in exactly the same way as what happens during a normal procedure call.

4.6.3 Optimization: Enabling a Privilege

The model of `enablePrivilege()` in section 4.4.2 differs somewhat from the Netscape implementation of stack inspection, where a stack frame F cannot successfully call `enablePrivilege(`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 JDK 1.2's implementation is closer to our model.

4.6.4 Optimization: Frame Credentials

Java implementations do not treat stack frames or their code as separate principals. Instead, they only track the public key which signed the code and call this the frame's principal. As we saw in section 4.1, for any stack frame, we can prove the stack frame speaks for the public key which 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 which signed that frame's code. This means that Java implementations do not have an internal notion of the frame credentials used here.

5 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 belief sets can be represented by a finite 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.

5.1 Belief Sets and Automata

We can simplify the representation of belief sets 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.

It follows that without affecting the result of the decision procedure we can rewrite each belief into a canonical form in which each atomic principal appears at most once, and the atomic principals appear in some canonical order. After rewriting the beliefs into canonical form, we can discard any duplicate beliefs from the belief set.

Since the set of principals is finite, and the set of targets is finite, and no principal or target may be mentioned more than once in a canonical-form belief, there is a finite set of possible canonical-form beliefs. It follows by a simple argument that only a finite number of canonical-form belief sets may exist.

We can therefore represent the evolution of a stack frame's belief set by a finite automaton. 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 `enablePrivilege()`, `disablePrivilege()` and `revertPrivilege()` correspond to state transitions[4].

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.

5.2 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 encodes the security state (the finite-state representation of the belief set) of the procedure's stack frame. This eliminates the need to have a separate pushdown stack for security states. We dub this approach *security-passing style*, by analogy to continuation-passing style [18], a transformation technique used by some compilers that also replaces an explicit pushdown stack with implicitly-passed procedure arguments.

We note that security-passing style can be implemented by rewriting code as it is being loaded into the system, to add the extra parameter to all procedures and procedure calls, and to rewrite the privilege-manipulation operations into equivalent operations on the security state. This is straightforward to implement for Java bytecode, since the bytecode format contains enough information to make rewriting possible.

[4] One more nicety is required. To implement `revertPrivilege()`, we need to remember what the security state was when each stack frame was created. We can encode this information in the finite state, or we can store it on the stack by doing another `push` operation on procedure call.

The main advantage of security-passing style is that once a program has been rewritten, it no longer needs any special security functionality from the JVM. The rewritten program consists of ordinary Java bytecode, which 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.

We are currently implementing security-passing style by rewriting bytecode at load time using the JOIE [3] tool. The rewriter is a trusted module which we add to the JVM.

A full description of security-passing style and its implications for programming language implementations will appear in a future paper.

6 Remote Procedure Calls

Another advantage of security-passing style is that it suggests an implementation strategy for remote procedure call (RPC) security. Though a simple translation of security-passing style into the RPC case does not work, security-passing style with a few modifications works well for RPCs.

RPC security has received a good deal of attention in the literature. The two prevailing styles of security are capabilities and access control lists [19, 5, 8, 16, 20]. Most of these systems support only simple principals. Even in systems that support more complex principals [22], the mechanisms to express those principals are relatively unwieldy.

This section discusses how to extend the Java stack inspection model across RPCs. One of the principal uses for ABLP logic is in reasoning about access control in distributed systems, and we use the customary ABLP model of network communication to derive a straightforward extension of our model to the case of RPC.

6.1 Channels

When two machines establish an encrypted channel between them, each machine proves that it knows a specific private key which corresponds to a well-known public key. When one side sends a message through the encrypted channel, we model this (following [1] and [22]) as a statement made by the sender's session key: we write K **says** s, where K is the sender's session key and s is the statement. As discussed in section 4.1, the public-key infrastructure and the session key establishment protocol together let us establish that K speaks for the principal that sent the message.

In order to extend Java stack inspection to RPCs, each RPC call must transmit the belief set of the RPC caller to the RPC callee. Since each of the caller's beliefs is sent through a channel established by the caller's virtual machine, a belief B of the caller's frame arrives on the callee side as K_{CVM} **says** B, where K_{CVM} is a cryptographic key that speaks for the caller's virtual machine. The stack frame that executes the RPC on the callee is given an initial belief set consisting of all of these arriving statements.

Note that this framework supports the intuition that a remote caller should not be allowed to access resources unless that caller's virtual machine is trustworthy. All of the beliefs transmitted across the network arrive as statements of the caller's virtual machine (or more properly, of its key); the callee will disbelieve these statements unless it trusts the caller's virtual machine.

This strategy fits together well with security-passing style. We can think of the transmitted belief set as a representation of the caller's security state: to pass a security state across the net we translate it into a belief set in canonical form; on arrival at the destination we translate it back into a security state.

There is one more issue to deal with. The RPC caller's belief set is expressed in terms of the caller's stack frames; though these are the "correct" beliefs of the caller, they are not useful to the callee, since the callee does not know about caller-side stack frames. To address this issue, before the caller sends a belief across the network, the caller replaces each stack-frame principal F_i with an encryption-key principal K_i such that $F_i \Rightarrow K_i$. K_i can be the key that signed F_i's code. If F_i was running unsigned code, then F_i is powerless anyway so beliefs regarding its statements can safely be discarded.

Figure 3 presents an example of how this would work. The Java stack inspection algorithm executes on the callee's machine when an access control decision must be made, exactly as in the local case.

6.2 Dealing with Malicious Callers

An interesting question is what an attacker can accomplish by sending false or misleading statements across a channel. If the caller's virtual machine is malicious, it may send whatever "beliefs" it wants, provided that they have the correct format. Regardless of the beliefs sent, each belief arrives at the callee as a statement made by the caller's

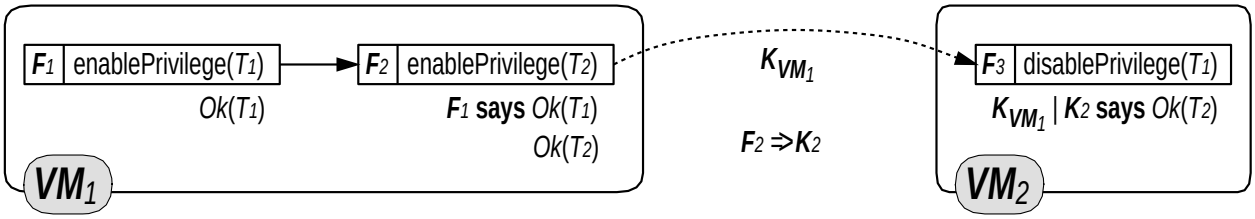

Figure 3: Example of interaction between stack frames via remote procedure call. Each rectangle represents a stack frame. Each stack frame is labeled with its name and its belief set (after its call to `enablePrivilege()` or `disablePrivilege()`). The larger rounded rectangles represent separate Java virtual machines, and the dotted arrow represents the channel used for a remote procedure call.

virtual machine. If the callee does not trust the caller, such statements will not convince the callee to allow access.

Suppose a malicious caller's virtual machine *MC* wants to cause an access to target *T* on some callee. The most powerful belief *MC* can send to support this attempt is simply *Ok*(*T*)[5]; this will arrive at the callee as *MC* **says** *Ok*(*T*). Note that this is a statement that *MC* can make without lying, since *MC* is entitled to add *Ok*(*T*) to its own belief set. Any lie that *MC* can tell is less powerful than this true statement, so lying cannot help *MC* gain access to *T*. The most powerful thing *MC* can do is to ask, under its own authority, to access *T*.

6.3 Dealing with Malicious Code on a Trustworthy Caller

Malicious code on a trustworthy caller also does not cause any new problems. The malicious code can add *Ok*(*T*) to its belief set, and that belief will be transmitted correctly to the callee. The callee will then allow access to *T* only if it trusts the malicious code to access *T*. This is the same result that would have occurred had the malicious code been running directly on the callee. This matches the intuition that (with proper use of cryptography for authentication, confidentiality, and integrity of communication) we can ignore machine boundaries if the communicating processes trust each other and the platforms on which they are running.

7 Conclusion

Commercial Java applications often need to execute untrusted code, such as applets, within themselves. In order to allow sufficiently expressive security policies, granting different privileges to code signed by different principals, the latest Java implementations now support a runtime mechanism to search the call-stack for code with different privileges and decide whether a given call-stack configuration is authorized to access a protected resource.

This paper has presented a formalization of Java's stack inspection using a logic developed by Abadi, Burrows, Lampson, and Plotkin [1]. Using this model, we have demonstrated how Java's access control decisions correspond to proving statements in ABLP logic. We have reduced the stack inspection model to a finite pushdown automaton, and described how to implement the automaton efficiently using security-passing style. We have also extended our model to apply to remote procedure calls and we have used the ABLP expression of this model to suggest a novel implementation for a Java-based secure RPC system. While the implementation of such an RPC system is future work, our model gives us greater confidence that the system would be both useful and sound.

[5]Technically, *MC* could send the belief `false`, which is even stronger; but we assume the protocol for transmitting beliefs will not allow this.

8 Acknowledgments

Thanks to Martín Abadi, Andrew Appel, Dirk Balfanz, Drew Dean and the anonymous referees for their comments and suggestions on this work and our presentation of it. Andrew Appel coined the term "security-passing style," convinced us of the importance of that technique, and suggested some of the state-machine implementation ideas.

Our work is supported by donations from Intel, Microsoft, Sun Microsystems, Bellcore, and Merrill Lynch. Edward Felten is supported in part by an NSF National Young Investigator award and an Alfred P. Sloan Fellowship.

References

[1] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. D. A calculus for access control in distributed systems. *ACM Transactions on Program-*

ming Languages and Systems 15, 4 (Sept. 1993), 706–734.

[2] BIRRELL, A. D., NELSON, G., OWICKI, S., AND WOBBER, E. P. Network objects. *Software: Practice and Experience S4*, 25 (Dec. 1995), 87–130.

[3] COHEN, G., CHASE, J., AND KAMINSKY, D. Automatic program transformation with JOIE. In *Proc. 1998 Usenix Technical Symposium* (June 1998). To appear.

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

[5] GONG, L. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (Oakland, California, May 1989), pp. 56–63.

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

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

[8] HU, W. *DCE Security Programming*. O'Reilly & Associates, Inc., Sebastopol, California, July 1995.

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

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

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

[12] MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P., AND LEVIN, M. I. *LISP 1.5 Programmer's Manual*, 2nd ed. The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1962.

[13] MCGRAW, G., AND FELTEN, E. W. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, New York, New York, 1997.

[14] MICROSOFT CORPORATION. *Trust-Based Security for Java*. Redmond, Washington, Apr. 1997. `http://www.microsoft.com/java/security/jsecwp.htm`.

[15] NETSCAPE COMMUNICATIONS CORPORATION. *Introduction to the Capabilities Classes*. Mountain View, California, Aug. 1997. `http://developer.netscape.com/library/documentation/signedobj/capabilities/index.html`.

[16] OBJECT MANAGEMENT GROUP. *Common Secure Interoperability*, July 1996. OMG Document Number: orbos/96-06-20.

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

[18] STEELE, G. L. Rabbit: a compiler for Scheme. Tech. Rep. AI-TR-474, MIT, Cambridge, MA, 1978.

[19] TANENBAUM, A. S., MULLENDER, S. J., AND VAN RENESSE, R. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems* (Cambridge, Massachusetts, May 1986), pp. 558–563.

[20] VAN DOORN, L., ABADI, M., BURROWS, M., AND WOBBER, E. Secure network objects. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (Oakland, California, May 1996).

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

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

A ABLP Logic

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 Java stack inspection.

Axioms About Statements

If s is an instance of a theorem of propositional logic then s is true in ABLP. (8)

If s and $s \supset s'$ then s'. (9)

$(A \textbf{ says } s \wedge A \textbf{ says } (s \supset s')) \supset A \textbf{ says } s'$. (10)

If s then A **says** s for every principal A. (11)

Axioms About Principals

$(A \wedge B) \textbf{ says } s \equiv (A \textbf{ says } s) \wedge (B \textbf{ says } s)$ (12)

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

$A = B \supset (A \textbf{ says } s \equiv B \textbf{ says } s)$ (14)

$\mid$ is associative. (15)

$\mid$ distributes over $\wedge$ in both arguments. (16)

$(A \Rightarrow B) \equiv (A = A \wedge B)$ (17)

$(A \textbf{ says } (B \Rightarrow A)) \supset (B \Rightarrow A)$ (18)

B Proofs

This section proves the theorems from section 4.5.

Theorem 1 (Termination) *The decision procedure always terminates.*

Proof: The result follows directly from the fact that $\mathcal{E}_F$ has bounded cardinality. 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. ∎

Theorem 2 (Soundness) *If the 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 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

   $$\mathcal{E}_F \supset (T \mid T \mid \cdots \mid T \textbf{ says } Ok(T)). \quad (19)$$

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

Theorem 3 (Equivalence to Stack Inspection) *The decision procedure described in section 4.5 is equivalent to the Java stack inspection algorithm of section 2.*

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 `enablePrivilege()` and `checkPrivilege()` primitives.

Our induction is over the number of *steps* taken, where a step is either a procedure call or an `enablePrivilege()` operation. For clarity, we ignore the existence of `disablePrivilege()`, `revertPrivilege()`, and 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 belief set is empty. In this base case, `checkPrivilege()` will fail in both systems.

Inductive step: We assume that N steps have been taken ($N \geq 0$) and we are in a situation where any `checkPrivilege()` call would yield the same result in both models. Now there are two cases:

enablePrivilege(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 belief set.

If this is followed by a `checkPrivilege(`T`)` operation, the operation will succeed in both systems, because of the new stack annotation or the new belief.

If it is followed by `checkPrivilege(`U`)` with $U \neq T$, the new stack annotation or belief will be irrelevant, 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 belief set.

If `checkPrivilege(`T`)` now occurs, there are two sub-cases. In the first sub-case, 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 decision procedure will deny access because every belief starts with "P **says**" and P does not speak for T.

In the second sub-case, 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 belief 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. ∎