Chapter 22

# IMPLEMENTATION AND VERIFICATION OF PROGRAMMABLE SECURITY

Stephen Magill, Bradley Skaggs, Mauricio Papa and John Hale

**Abstract** This paper presents a methodology for augmenting programming languages with configurable security services. In particular, it describes JPAC, a Java extension that provides syntax for expressing discretionary package-based access control. Access control is based on a variation of a ticket-based authorization model. The authorization model has been succesfully implemented in the Isabelle theorem proving environment. Moreover, a novel cryptographic verification formalism is used to analyze JPAC's secure method invocation protocol. Programmable security architectures and cryptographic protocol verification formalisms used in concert can provide verifiably secure programming systems for Internet applications.

**Keywords:** Formal methods, programmable security, authentication, process calculi

## 1. Introduction

Internet computing is a catalyst for the development of new programming language protection models, security APIs, and software integration and interoperability frameworks. Developers rely on protection models to check code integrity and guard memory boundaries at compile-time and run-time [4, 8]. Together, protection models and security APIs comprise the state of the art for safe-guarding applications running in open, heterogeneous environments. However, these tools do not ensure that a security policy articulated with an API is consistent or viable. Moreover, very little is available to programmatically link elements in a protection model with a security API. As a result, security APIs are commonly used in an *ad hoc* fashion yielding unpredictable security policies.

Programmable security provides syntactic and semantic constructs in programming languages for coherently embedding security functionality within applications [9]. Developers use special syntax to express security policies

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35697-6_26

E. Gudes et al. (eds.), *Research Directions in Data and Applications Security*

within code in the same way that types are used to express constraints on variable behavior. This approach facilitates compile-time and run-time security-checking (analogous to type-checking) to verify that no potential security policy violations exist within a program.

This paper describes the Java Package Access Control (JPAC) System. JPAC extends the Java programming language with syntax for specifying discretionary access control policies at the package level. The JPAC access control scheme is based on a primitive authorization model that has been mechanized in Isabelle, an interactive theorem prover. Access to Java data and code elements under JPAC is fully mediated by a secure method invocation scheme. As this paper demonstrates, both the authorization model and its implementation using the secure method invocation scheme are amenable to formal analysis.

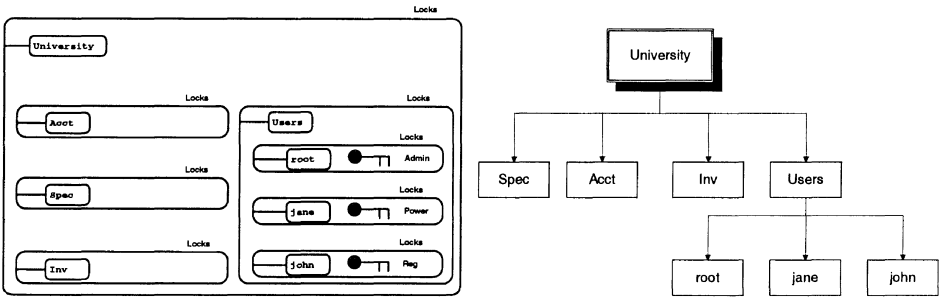
Section II of the paper discusses the broad topic of programmable security. Section III presents our authorization model and illustrates the potential for formal analysis of security policies expressed in the model. Section IV describes the JPAC archtitecture and each of its essential components. Section V provides a detailed examination of the secure method invocation protocol, and section VI presents a formal analysis of the scheme using a novel cryptographic protocol verification system. Section VII provides conclusions.

## 2. Programmable Security

Object-oriented programming languages employ protection schemes based on classes, variables and methods. Java 1.0 provides packages to group program units (classes and interfaces), creating access boundaries [4]. Java 1.2 lets developers define protection domains to specify sets of classes sharing identical permissions [7, 8]. The added functionality is given in an API. Wallach *et al.* propose extensions to the Java security model that employ capabilities and namespace management techniques [24]. Java capabilities are implemented based on the fact that references to objects cannot be fabricated due to Java's type safety features. The disadvantage of these approaches is that no significant compile-time security checking can be performed.

Early work in [5] describes a compile-time mechanism to certify that programs do not violate information flow policies, while [2] provides a flow logic to verify that programs satisfy confinement properties.

In [23], Volpano *et al.* recast the information flow analysis model in [5] within a type system to establish its soundness. This work led to a sound type system for information flow in a multi-threaded language [20]. JPAC differs in that it promotes a foundational authorization model as a common substrate for various access control schemes [9] to support the static analysis of secure program interoperability.




(a) Physical (b) Hierarchical

*Figure 1.* System views.

Van Doorn *et al.* extend Modula-3 network objects with security features in [22]. Secure network objects (SNOs) bind programming languages into service for integrating security into objects and methods. The SNO runtime design most closely resembles ours (e.g., using capabilities and access control lists), but the SNO syntax extensions express operational semantics as opposed to policy specifications.

Myers and Liskov describe a decentralized information flow control model in [17]. Myers implemented these ideas in JFlow [18]. While JFlow uses syntactic extensions in a dialect of Java to express information flow, our languages capture more abstract security policies in the discretionary access control domain.

The SLam calculus is a typed $\lambda$-calculus that tracks relevant security properties of programming elements [12]. A compiler that executes static checks enforces type system rules to guarantee program security. While the SLam calculus and our approach are both amenable to static analysis, their formalism relies heavily on secure type systems in functional languages.

## 3. Authorization Model

This section presents a formal semantics for the ticket-based authorization model and a simple scenario illustrating its use and capabilities. A system in our model is represented by a hierarchical structure of node components. Tickets embedded in messages as unforgeable tokens are analogous to capabilities [13, 14], conveying privileges of message originators. Message passing only occurs directly between two adjacent object nodes.

To illustrate the concept, consider a `University` with a transactions system consisting of four databases: `Spec`, `Inv`, `Acct` and `Users` containing equipment specifications, inventory, account information and users, respectively (see Figure 1a). Furthermore, `Users` contains a set of three subjects

| Predicate | Note |
|---|---|
| $Parent\ s\ o_1\ o_2\ (o_1 \neq o_2)$ | $o_1$ is *Parent* of $o_2$ in state $s$ |
| $Adj\ s\ o_1\ o_2 = Parent\ s\ o_1\ o_2\ \vee\ Parent\ s\ o_2\ o_1$ | $o_1$ is adjacent to $o_2$ in state $s$ |
| $Key\ s\ o_1\ t$ | $o_1$ has *Key* named $t$ in state $s$ |
| $Lock\ s\ o_1\ o_2\ t\ (Adj\ s\ o_1\ o_2)$ | $o_1$ has lock named $t$ on $o_2$ in state $s$ |
| $Match\ s\ o_1\ o_2\ o_3\ = \exists t.\ Key\ s\ o_1\ t\ \wedge\ Lock\ s\ o_2\ o_3\ t$ | Key/Lock matching in state $s$ |
| $Access\ s\ o_1\ o_1$ | Objects have access to themselves |
| $Access\ s\ o_1\ o_2\ \wedge\ Match\ s\ o_1\ o_2\ o_3\ \Rightarrow\ Access\ s\ o_1\ o_3$ | $o_1$ has access to $o_3$ |

*Figure 2.* Authorization model.

where each subject possesses a key indicating its access level: `root` (an administrator with read, write and query permissions), `john` (a regular user with read permission) and `jane` (a power user with read and write permission).

Our model consists of rules defining the hierarchy, local access level and access level in the hierarchy as described in the following subsections.

**Hierarchy Definition** The structure of the hierarchy is represented by a set of predicates of the form $Parent\ s\ o_1\ o_2\ (o_1 \neq o_2)$ [Rule 1], indicating that $o_1$ is the parent node of $o_2$ in state $s$. Preconditions for the predicates are in parentheses. The hierarchy in our example, as shown in Figure 1b, is represented by the following set of seven predicates:

| | | |
|---|---|---|
| *Parent s* `University Spec` | *Parent s* `University Inv` | *Parent s* `University Acct` |
| *Parent s* `University Users` | *Parent s* `Users root` | *Parent s* `Users jane` |
| | *Parent s* `Users john` | |

The adjacency relationships between objects, as required by the message-passing scheme, can be formally specified with the following predicate [Rule 2]: $Adj\ s\ o_1\ o_2 = Parent\ s\ o_1\ o_2\ \vee\ Parent\ s\ o_2\ o_1$. This auxiliary predicate is used to establish authorized message flow. Messages can only be directly passed between two adjacent nodes.

**Local Access Level** Conceptually, tickets represent *keys* held by subjects that match *locks* held by objects. Keys are checked for matching object locks to authorize access requests.

Access level in the hierarchy is defined by predicates of the form $Key\ s\ o_1\ t$ [Rule 3] and $Lock\ s\ o_1\ o_2\ t\ (Adj\ s\ o_1\ o_2)$ [Rule 4]. At this level, the model mandates that $o_1$ and $o_2$ be adjacent for $o_1$ to hold such a lock. Access to non-adjacent nodes is determined by inferring access level as described in the next subsection. The first predicate is true when $o_1$ has a key named $t$ in state $s$ and the second when $o_1$ has a lock named $t$ on object $o_2$ in state $s$ (and the precondition is satisfied). Container nodes must be given the corresponding locks in order to validate/delegate messages:

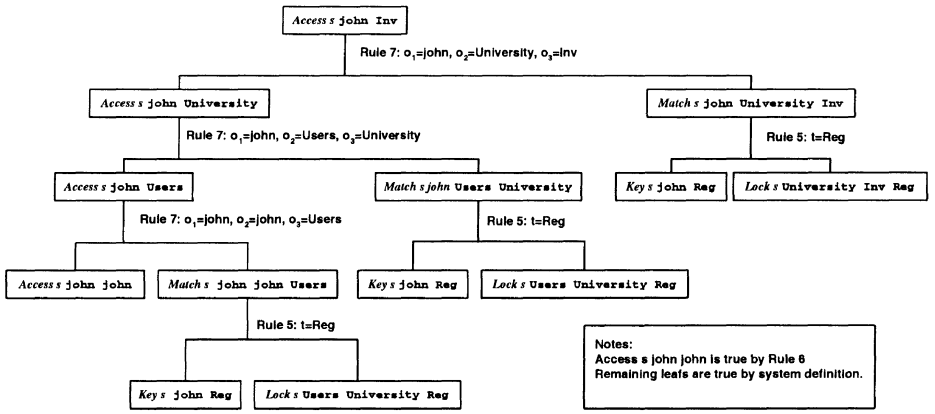



*Figure 3.* Backward chaining proof.

| | |
|---|---|
| *Lock s* `University Spec` *Admin* | *Lock s* `University Spec` *Power* |
| *Lock s* `University Inv` *Admin* | *Lock s* `University Inv` *Power* |
| *Lock s* `University Inv` *Reg* | *Lock s* `University Users` *Admin* |
| *Lock s* `University Users` *Power* | *Lock s* `University Users` *Reg* |

This embodies the security policy put in place by `University`. Similarly, a lock structure is defined for `Users`, having locks on each user corresponding to their access level and locks on `University` to check messages going up the hierarchy:

| | |
|---|---|
| *Lock s* `Users root` *Admin* | *Lock s* `Users jane` *Power* |
| *Lock s* `Users john` *Reg* | *Lock s* `Users University` *Admin* |
| *Lock s* `Users University` *Power* | *Lock s* `Users University` *Reg* |

Each user in the hierarchy also possesses locks to check appropriate permissions to send messages through `Users`:

*Lock s* `root Users` *Admin* | *Lock s* `jane Users` *Power* | *Lock s* `john Users` *Reg*

Finally, each user is given a key matching his/her access level:

*Key s* `root` *Admin* | *Key s* `jane` *Power* | *Key s* `john` *Reg*

**Object Access Level Inference** We first define a predicate $Match$, which expresses when a message has permission to be forwarded to $o_3$ on behalf of $o_1$ in $o_2$ (a key/lock matching operation in essence) [Rule 5]:

$$Match\ s\ o_1\ o_2\ o_3\ =\ \exists\, t.\, Key\ s\ o_1\ t\ \wedge\ Lock\ s\ o_2\ o_3\ t.$$

We define another predicate to represent the ultimate goal of a message. $Access\ s\ o_1\ o_2$ specifies that $o_1$ can access $o_2$ from its point of origin in state $s$. Now we can complete the formalization by creating an inductive definition for access between nodes in an object hierarchy with the addition of two rules:

```
theory auth=Main:
types name=nat
datatype Object= OB name "key list" "lock list" "Object
list"
and       lock = LOCK "name" "name"
and       key  = KEY  "name"
consts OName:: "Object =>  name"
       OKeys:: "Object => key list"
       OLocks:: "Object => lock list"
       OChildren:: "Object => Object list"
       LName:: "lock => name"
       LObject:: "lock => name"
       KName:: "key => name"
primrec "OName (OB name keys locks children) = name"
primrec "OKeys (OB name keys locks children) = keys"
primrec "OLocks (OB name keys locks children) = locks"
primrec "OChildren (OB name keys locks children) =
children"
primrec "LName (LOCK name1 name2) = name1"
primrec "LObject (LOCK name1 name2) = name2"
primrec "KName (KEY name) = name"
consts  OInStateO:: "Object => Object => bool"
        OInStateL:: "Object => Object list => bool"
primrec "OInStateO O1 (OB sname skeys slocks olist) =
((O1 = (OB sname skeys slocks olist))|(OInStateL O1
olist))"
        "OInStateL O1 [] = False"
        "OInStateL O1 (oh#ol) = ((OInStateO O1
oh)|(OInStateL O1 ol))"

constdefs Parent:: "Object => Object => Object => bool"

        "Parent O1 O2 State == ((OInStateO O1 State) &
(OInStateO O2 State) & (O2 mem (OChildren O1)))"

constdefs Adj:: "Object => Object => Object => bool"
        "Adj O1 O2 State == Parent O1 O2 State | Parent
O2 O1 State"

constdefs Key:: "Object => name => Object => bool"
        "Key O1 name State == (((KEY name) mem (OKeys
O1)) & (OInStateO O1 State))"

constdefs Lock:: "Object => Object => name => Object
=> bool"
        "Lock O1 O2 name State == ((Adj O1 O2 State) &
((LOCK name (OName O2)) mem (OLocks O1)))"

constdefs Match:: "Object => Object => Object => Object
=> bool"
        "Match O1 O2 O3 State == (? t .(Key O1 t
State)&(Lock O2 O3 t State))"

consts Access :: "(Object * Object * Object) set"

inductive "Access"
intros
 Base: "OInStateO O1 State==>(O1 , O1 , State): Access
"
 Ind:  "[| ? O2. ((O1 , O2 , State):Access & Match O1
O2 O3 State) |] ==> (O1 , O3 , State): Access"
```

*Figure 4.* Authorization model as implemented in the Isabelle theorem prover.

$Access\ s\ o_1\ o_1 = true$ [Rule 6] and $Access\ s\ o_1\ o_2 \wedge Match\ s\ o_1\ o_2\ o_3 \Rightarrow Access\ s\ o_1\ o_3$ [Rule 7].

The first rule indicates that objects always have access to themselves, and forms a base case for inductive access checking. The second provides the inductive step, stating that if $o_1$ can access $o_2$, and if $o_1$ holds a key matching a lock in $o_2$ for $o_3$, then $o_1$ can access $o_3$. The entire model is summarized in Figure 2.

Figure 3 shows a sketch of the backward-chaining proof showing that John has access to the inventory, i.e. $Access\ s$ `john Inv`, by using the formal model described in this section.

The authorization model has been used in a prototype implementation providing programmable security at the package level in the Java programming language and as a formal foundation for the authorization service in a coordination language for heterogenous objects.

**Authorization Model Verification** The model presented in this section has also been implemented in the Isabelle theorem proving environment (see Figure 4). The natural numbers are used to represent names (left-hand side of Figure 4, **types** clause). Three basic datatypes are created in the theory: `Object`, `lock` and `key` with constructors `OB`, `LOCK` and `KEY` respectively (mirroring the model). An `Object` is characterized by a name, a list of keys, a list of locks and a list of objects (used to represent the object hierarchy). Similarly, two names are used to characterize a `lock`: the name of the lock and the name of the object whose access will be controlled by this lock. A `key` is characterized by a name only.

The `consts` clauses define types for access methods to the individual components of each of the three datatypes. For instance, `OName` has type `Object => name` and returns the name of the object passed to the function. In our Isabelle model, the state *s* is represented by an `Object`. Consequently, we must define functions that will determine whether or not an object is part of a given state, i.e. whether or not it is part of the current object structure representing the state. The `Object` datatype definition is a typical case of nested recursion, it is defined in terms of a list of objects and a list is itself a recursive datatype. Determining membership of an object to the current state, requires the use of two mutually recursive functions: `OInStateO` and `OInStateL`. These two functions cover all possible variations of the input.

Having defined the basic datatypes and associated auxiliary functions, the rules in Figure 2 can be easily incorporated into Isabelle (see right-hand side of Figure 4). For instance, `Parent O1 O2 State` tests whether or not both `O1` and `O2` are in the state and that `O2` is a child of `O1` (by using the membership function defined in the list theory). All other rules are defined similarly with the exception of the `Access` rule. `Access` is represented by an inductively defined set of triples. A triple `(O1, O2, State)` is in `Access` if and only if the triple satisfies the induction rules `Base` and `Ind` which correspond to rules 6 and 7 in the authorization model. The theorem `(john, invt, university): Access` in Isabelle corresponds to the example previously proved by hand (Figure 3).

The outline for the proof in Figure 3 served as a guide to complete the backward-chaining proof in Isabelle.

## 4. Java Package Access Control

This section presents, a programmable package-based protection scheme for Java (JPAC) using the authorization model presented in the previous section. Standard Java syntax offers four different access level qualifiers for class members with well defined access boundaries: `private`, `protected`, `public` and `package`. In particular, the `public` qualifier is the only one that allows a programmer to cross the package boundary. Unfortunately, declaring a member `public` grants access to the world. JPAC uses minimal syntax extensions to provide developers with discretionary and fine-grained access control for Java applications. Note that JPAC extends, not replaces, the existing Java security architecture. Constructs on the extended grammar are translated into standard Java by means of a specialized preprocessor.

**JPAC Grammar** Java syntax extensions used to express package-based protection are shown in Figure 5. The EBNF productions in Figure 5 change the way a compilation unit is named by making `PackageDeclaration`

```
CompilationUnit:
    PackageDeclaration [ ImportDeclarations ] [ TypeDeclarations ]
  PackageDeclaration:
    package [ PackageName ] Properties
  Properties:
    protection { [ Property] }
  Property:
    PackageName ; ( PackageName )*
    guarded ;
    unsecured ;
```

```
package Acct
    protection {
  Users;}
class MyClass {
  ... }
```

```
package Inv
    protection {
  guarded;}
class MyClass {
  ... }
```

```
package Spec
    protection {
  unsecured;}
class MyClass {
  ... }
```

*Figure 5.* Extended syntax.

mandatory and adding a `Properties` production to it. Unnamed compilation units are specified by declaring a nameless package.

Three different access modes are considered in our approach: list-based, guarded and unsecured. List-based access provides the ability to specify which packages will be granted rights and permissions. All classes in the access list must be JPAC protected (checked at compile time). Guarded access provides the ability to grant access to all JPAC classes. Unsecured is used for integrating JPAC and legacy code. Three examples of legal compilation units can be found at the bottom of Figure 5.

JPAC program elements are organized into an object hierarchy. A root object resides at the top of the hierarchy, below it are objects modeling packages, classes and instances. All protected packages are provided with a message handler; a class specifically designed to provide messaging facilities. Access requests are carried by messages that flow from the message handler of the package representing the request source to the message handler of the destination package.

The JPAC semantics are derived from the ticket-based authorization scheme and object hierarchy described in Section 3.

## 5. Secure Method Invocation

This section presents a secure method invocation protocol used to implement the authorization model described in Section 22.3. It also provides a formal model for proving relevant security properties.

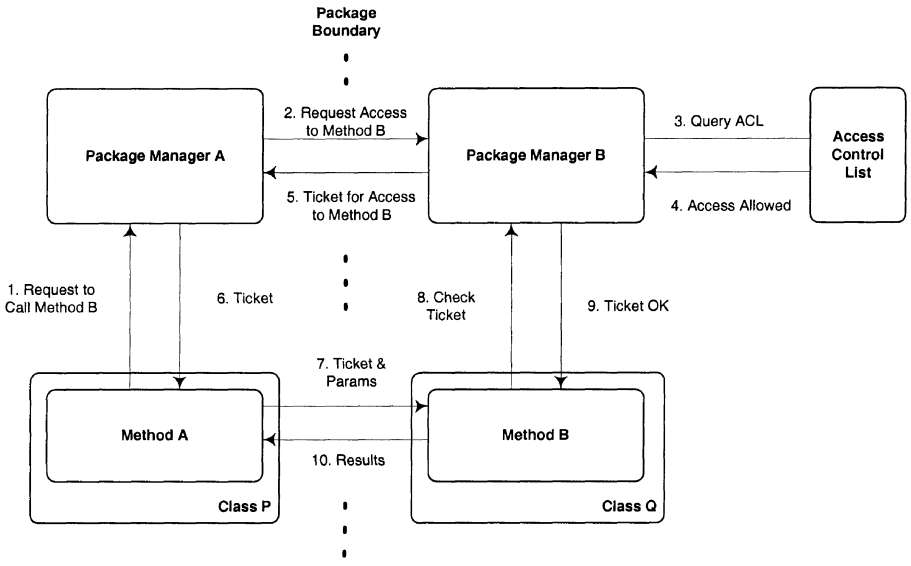



*Figure 6.* Secure method invocation.

The secure method invocation protocol uses a ticket-based access control scheme to enforce application-level policies at runtime. Authenticated entities obtain a ticket which can be exchanged for method access. Most of the authentication work rests with package managers, so that methods remain relatively lightweight in JPAC.

An overview of the protocol is provided in Figure 6. The protocol consists of ten communication steps that are functionally divided into two phases: obtaining a ticket from the callee's package manager and the actual method call.

This two-phase scheme has the advantage of reducing system overhead. Once a ticket is obtained, successive calls proceed by skipping the first phase.

## 6. Verification

This section demonstrates the use of a novel cryptographic protocol verification formalism to analyze JPAC's secure method invocation protocol. The formalism uses elements from various process calculi [1, 15, 16]. It combines the convenient modeling properties of process calculi with the powerful reasoning ability of authentication logics, such as BAN [3]. In the analysis process, a subtle flaw in the secure method invocation protocol is discovered and corrected.

**Protocol Logic** The secure method invocation protocol is formally modeled with a subset of the protocol logic defined in [19]. The logical system is a simple process calculus with pattern-based message exchange and support for

$$
\begin{array}{rcl}
key & ::= & K_n \mid K_n^{-1} \mid K_n^S \\
data & ::= & key \mid n \\
message & ::= & [m_1, m_2, ...m_i] \mid \\
 & & message_{key} \mid data \\
pattern & ::= & [p_1, p_2, ...p_i] \mid pattern_{key} \mid \\
 & & data \mid n? \\
comm & ::= & \rightarrow pattern\ comm \mid \\
 & & \hat{} \ message\ comm \mid \\
 & ::= & \#n.\ comm \mid \{\}
\end{array}
$$

(a) BNF Grammar

$$
\begin{array}{rcl}
n & \sim & n \\
[p_1, p_2, ...p_i] & \sim & [m_1, m_2, ...m_i] \ \ if\ \forall k \cdot p_k \sim m_k \\
pattern_{K_n^{-1}} & \sim & message_{K_n} \\
pattern_{K_n} & \sim & message_{K_n^{-1}} \\
pattern_{S_n} & \sim & message_{S_n} \\
n? & \sim & m \ \ (\forall m)
\end{array}
$$

(b) Pattern matching

*Figure 7.* Protocol logic.

both public key and symmetric encryption. The BNF grammar for the process calculus is shown in Figure 7(a).

$n$ denotes a member of an infinite set of names, which represents our data. A key is either public, private, or symmetric. The term "data" is used to encompass keys and names. Messages are strings of data or encrypted messages ($message_{key}$ denotes a message encrypted under $key$). Pattern structure mirrors message structure with the addition of a "wildcard" construction ($n?$), which makes $n$ act much like a variable.

$comm$ comprises the set of communication sequences, with output denoted by ($\hat{}$) and input denoted by ($\rightarrow$). The $\#n.\ comm$ construction is read as "new name $n$ in $comm$". It binds $n$ such that $n$ inside $comm$ is different from $n$ outside $comm$. The result is that $\#n.\ comm \cong comm[n'/n]$ provided $n'$ is not free in the system. A name appearing in the system is bound if it is a wildcard or new name and free otherwise.

An agent is defined by a unique ID and a communication sequence. Agent IDs are considered public knowledge. A system is a group of concurrent agents. Concurrency is denoted by (&), a commutative operator. Addition or deletion of an agent whose communication is $\{\}$ (the nil agent) also results in a system congruent to the original (a simplification rule).

$$
\begin{array}{rcl}
agent & ::= & (n, comm) \\
system & ::= & agent\ \&\ system \mid \{\}
\end{array}
$$

Communication occurs when the pattern exposed by one agent matches the message offered by another (pattern matching rules are shown in Figure 7(b)). At this point, the message is transmitted and both agents are said to reduce. The following reduction rules define this behavior:

$$
\textbf{Out:}\ \hat{}\ m\ a \stackrel{\overline{m}}{\Rightarrow} a \qquad \textbf{In:}\ \frac{m \sim p}{\rightarrow p\ a \stackrel{m}{\Rightarrow} a[m/p]} \qquad \textbf{Comm:}\ \frac{a_1 \stackrel{\overline{m}}{\Rightarrow} a_1' \ \ a_2 \stackrel{m}{\Rightarrow} a_2' \ \ a_1, a_2 \in S}{S \Rightarrow S[a_1'/a_1][a_2'/a_2]}
$$

The **Out** and **In** rules define agent behavior for an agent offering a message and an agent exposing a pattern respectively. The **Comm** rule defines agent communication. Operators $\Rightarrow$, $\overset{\overline{m}}{\Rightarrow}$ and $\overset{m}{\Rightarrow}$ are used to indicate system reduction, agent reduction with output $m$ and agent reduction with input $m$.

**Formal Specification** In any run of the secure method invocation protocol, there are four agents participating: the initiator method, initiator package, receiver method and receiver package. The definitions for these four agents will be abstracted over the agent IDs. The convention adopted in the formalism below gives IDs much the same properties as IP addresses. $\mathrm{P_{enc}}$ can be thought of as the "ticket" which grants access to Method B.

The package manager and the package managed by it are assumed to be running on the same machine. Thus, communication within a package is considered to be secure. However, they are included here for completeness. Since the agent ID is evident from our function naming convention it has been omitted in following definitions; $\mathrm{agent}_n$ is associated with ID $n$.

$$
\begin{aligned}
agent_{m_A}(m_A, m_B, A, B) &= {}^{\wedge}\,[A, m_B, P] \rightarrow [m_A, P_{enc}?]\,{}^{\wedge}\,[m_B, P_{enc}] \\
&\quad \rightarrow [m_A, R_{enc}?]\,{}^{\wedge}\,[A, R_{enc}] \rightarrow [m_A, R?] \\
agent_{m_B}(m_A, m_B, A, B) &= \rightarrow [m_B, P_{enc}?]\,{}^{\wedge}\,[B, P_{enc}] \rightarrow [m_B, P?] \\
&\quad {}^{\wedge}\,[B, R, P_{enc}] \rightarrow [m_B, R_{enc}?]{}^{\wedge}[m_A, R_{enc}] \\
agent_A(m_A, m_B, A, B) &= \rightarrow [m_A, m_B, P?]\,{}^{\wedge}\,[B, A, [m_B, P]_{K_A^{-1}}] \rightarrow [A, [m_B, P_{enc}?]_{K_A^{-1}}] \\
&\quad {}^{\wedge}\,[m_A, P_{enc}] \rightarrow [A, [m_B, [R?, N?, P]_{K_B}]_{K_A{}^{-1}}]{}^{\wedge}\,[m_A, R] \\
agent_B(m_A, m_B, A, B) &= \rightarrow [B, A, [m_B, P?]_{K_A}]\,\#N.\,{}^{\wedge}\,[A, [m_B, [N, P]_{K_B^{-1}}]_{K_A}] \\
&\quad \rightarrow [B, [N, P]_{K_B}]\,{}^{\wedge}\,[m_B, P] \rightarrow [B, R, [N, P]_{K_B}] \\
&\quad {}^{\wedge}\,[m_B, [m_B, [R, N, P]_{K_B^{-1}}]_{K_A}]
\end{aligned}
$$

R is the result of the method call and P is the parameter set. Notice that pattern matching is used to check the nonce in $\mathrm{agent_B}$.

**Security and Knowledge Inference** Let $\mathrm{CS}(S, S')$ denote the set of all messages passed during all reductions of $S$ to $S'$, hereafter referred to as the communications set of $(S, S')$.

To analyze the security of a protocol, the formalism must examine what knowledge an intruder could collect during protocol runs and whether this knowledge could be used to gain access to private information. All communications are assumed to be over open channels. This allows the intruder to see every message exchanged in the protocol run.

$\mathrm{Kn}(M)$, the knowledge gained from the set of messages $M$, is defined as:

$$\frac{m \in M}{m \in \mathrm{Kn}(M)} \qquad \frac{m, ml \in \mathrm{Kn}(M)}{m::ml \in \mathrm{Kn}(M)}$$

$$\frac{m::ml \in \mathrm{Kn}(M)}{m \in \mathrm{Kn}(M) \wedge ml \in \mathrm{Kn}(M)} \qquad \frac{m \in \mathrm{Kn}(M) \wedge k \in \mathrm{Kn}(M)}{\{m\}_k \in \mathrm{Kn}(M)}$$

$$\frac{\{m\}_{k_1} \in \mathrm{Kn}(M) \wedge k_2 \in \mathrm{Kn}(M) \wedge k_1 \sim k_2}{m \in \mathrm{Kn}(M)}$$

where $k_1 \sim k_2$ means "$k_1$ matches $k_2$" in the sense that a public key matches its corresponding private key and two symmetric keys match each other if they are the same key. The intruder's knowledge after the system changes from state $S$ to state $S'$ then becomes $\mathrm{Kn}(CS(S, S'))$.

It is now possible to define the meaning of a secure system. Let:
$T(S) = \{(S, S') \mid S \stackrel{*}{\Rightarrow} S'\}, S + I = S \cup \{\hat{} \; m \mid m \in \mathrm{Kn}(CS(T(S)))\} \cup \{\rightarrow n?\}$ *and* secure $(S, \mathrm{M_p}) = \nexists m \in \mathrm{Kn(CS(T(S + I)))} \cdot m \in \mathrm{M_p}$ *where* $\mathrm{M_p} =$ *set of messages that should remain private*

A system is secure if the intruder cannot gain access to elements in $\mathrm{M_p}$. The use of S + I in the definition of "secure" acknowledges that the intruder can take an active approach and insert or remove messages from the communications channel.

Protocol security can now be characterized. A protocol is a set of functions of type $I \rightarrow agent$ where $I = \prod_\alpha IDs$. $\alpha$ is a finite indexing set and $IDs$ is the subset of $names$ that represent agent IDs. Given a protocol, $P$, security of this protocol with respect to a set of private messages is defined as:

$$\text{Let S(P)} = \bigcup p(I) \;\; (p \in P), \text{ secure } (P, M_p) = secure(S(P), M_p)$$

S is the union of the co-domains of the protocol functions. The security property for protocols, as given above, states that a protocol is secure if all the states that are reachable using the communication patterns defined by that protocol are secure. Despite the apparent complexity of the task, a relatively straightforward inductive approach suffices as a proof strategy. The base case consists of a system whose only requirements are that every agent is an application of a protocol function and every element of $\mathrm{M_p}$ is present in the system. It eventually must be shown that:

$$\forall \mathrm{p} \in \mathrm{P} \cdot \text{secure } (S, M_p) \Longrightarrow \text{secure } (p(I) \& S, M_p)$$

In our analysis we consider concurrent runs of a protocol and the intruder's ability to communicate with other agents. As a rule, we keep track of the intruder's knowledge ($\mathrm{Kn_I}$) at all times. If it is learned that a particular protocol function adds to $\mathrm{Kn_I} \cap Kn(M_p)$, then that line of communications is followed, as it may lead to the discovery of a security vulnerability.

**Protocol Analysis** This approach has been used to reveal and correct a flaw in the secure method invocation protocol. Since the goal of the protocol is that only an authorized methods get results of a method call, our goal was to show that the protocol is secure with respect to the method result. Initial analysis showed that an intruder could access method results by using a package manager as an oracle.

As a result, the final version of the protocol includes a nonce generated by the caller to prevent the intruder from gaining access to method results unless it is authorized. Based on this analysis, the new protocol is secure.

## 7. Conclusions

Programmable security and programmable access control allow developers to express verifiable protection policies in their applications. JPAC extends the Java language with syntax for expressing package-level discretionary policies. JPAC classes and interfaces can be seamlessly integrated within native Java applications, allowing developers to customize protection policies for selected software components.

A dual complement of formal methods has been used to support the design of JPAC's authorization model and runtime access control mechanisms. The ticket-based authorization model was mechanized in a simple logic using Isabelle to demonstrate the potential for static and dynamic access control policy checking. JPAC's secure method invocation scheme was subjected to formal analysis via a cryptographic protocol verification system, leading to the discovery and elimination of a flaw in the original protocol. The collective success of these two efforts strongly suggests an expansive role for formal methods and tools in the development of programmable security infrastructures.

## Acknowledgments

This research was supported by MPO Contract MDA 904-98-C-A900 and NSF Grant CCR-9984774.

## References

[1] Abadi, M. and Gordon D., Reasoning about cryptographic protocols in the Spi calculus, *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pp. 36-47, 1997.

[2] Andrews, G. and Reitman, R., An axiomatic approach to information flow in programs, *ACM Transactions on Programming Languages and Systems*, vol. 2(1), pp. 56-76, 1980.

[3] Burrows, M., Abadi, M. and Needham, R., A logic of authentication, *ACM Transactions on Computer Systems*, vol. 8(1), pp. 18-36, 1990.

[4] Dean, D., Felten, E. and Wallach, D., Java security: ¿From HotJava to Netscape and beyond, *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 190-200, 1996.

[5] Denning, D. and Denning, P., Certification of programs for secure information flow, *Communications of the ACM*, vol. 20(7), pp. 504-513, 1977.

[6] Gilgor, V., Huskamp, J., Welke, S., Linn, C. and Mayfield, W., Traditional capability based systems: An analysis of their ability to meet the trusted computer security evaluation criteria, IDA Paper P-1935, Institute for Defense Analyses, Alexandria, Virginia, 1987.

[7] Gong, L., Mueller, M., Prafullchandra, H. and Schemers, R., Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2, *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pp. 103-112, 1997.

[8] Gong, L. and Schemers, R., Implementing protection domains in the Java Development Kit 1.2, *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, pp. 125-134, 1998.

[9] Hale, J., Threet, J. and Shenoi, S., Capability-based primitives for access control in object-oriented systems, in *Database Security, XI: Status and Prospects* (eds. T.Y. Lin and X. Qian), Chapman and Hall, London, pp. 134-150, 1998.

[10] Hale, J., Papa, M. and Shenoi, S., Programmable security for object-oriented systems, in *Database Security, XII: Status and Prospects* (ed. S. Jajodia), Kluwer, Dordrecht, The Netherlands, pp. 109-126, 1999.

[11] Hale, J., Threet, J. and Shenoi, S., A ticket based access control architecture for object systems, *Journal of Computer Security*, vol. 8, pp. 43-65, 2000.

[12] Heintze, N. and Riecke, J., The SLam calculus: Programming with security and integrity, *Proceedings of the Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pp. 365-377, 1998.

[13] Karger, P., An augmented capability architecture to support lattice security, *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 2-12, 1984.

[14] Karger, P., Implementing commercial data integrity with secure capabilities, *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 130-139, 1988.

[15] Milner, R., *Communication and Concurrency*, Prentice-Hall, New York, 1989.

[16] Milner, R., Parrow, J. and Walker, D., A calculus of mobile processes, Technical Report ECS-LFCS-89-85&86, University of Edinburgh, Edinburgh, U.K., 1989.

[17] Myers, A. and Liskov, B., A decentralized model for information flow control, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pp. 129-142, 1997.

[18] Myers, A., JFlow: Practical mostly-static information flow control, *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pp. 229-241, 1999.

[19] Papa, M., Bremer, O., Magill, S., Hale, J. and Shenoi, S., Simulation and analysis of cryptographic protocols, in *Data and Applications Security: Developments and Directions* (eds. B. Thuraisingham, R. van de Riet, K. Dittrich and Z. Tari), Kluwer, Dordrecht, The Netherlands, pp. 89-100, 2001.

[20] Smith, G. and Volpano, D., Secure information flow in a multi-threaded imperative language, *Proceedings of the Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pp. 355-364, 1998.

[21] Sun Microsystems, Clarifications and amendments to the Java language specification, www.java.sun.com/docs/books/jls/clarify.html, 1999.

[22] Van Doorn, L., Abadi, M., Burrows, M. and Wobber, E., Secure network objects, *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 211-221, 1996.

[23] Volpano, D., Smith, G. and Irvine, C., A sound type system for secure flow analysis, *Journal of Computer Security*, vol. 4(3), pp. 167-187, 1996.

[24] Wallach, D., Balfanz, D., Dean, D. and Felten, E., Extensible security architectures for Java, *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pp. 116-128, 1997.