CHAPTER 20

LANGUAGE EXTENSIONS FOR PROGRAMMABLE SECURITY

J. Hale, R. Chandia, C. Campbell, M. Papa and S. Shenoi

Abstract Software developers rely on sophisticated programming language protection models and APIs to manifest security policies for Internet applications. These tools do not provide suitable expressiveness for fine-grained, configurable policies. Nor do they ensure the consistency of a given policy implementation. Programmable security provides syntactic and semantic constructs in programming languages for systematically embedding security functionality within applications. Furthermore, it facilitates compile-time and run-time security-checking (analogous to type-checking). This paper introduces a methodology for programmable security by language extension, as well as a prototype model and implementation of JPAC, a programmable access control extension to Java.

Keywords: Cryptographic protocols, simulation, verification, logics, process calculi

1. INTRODUCTION

Internet computing is a catalyst for the development of new programming language protection models and security APIs. Developers rely on protection models to check code integrity and guard memory boundaries at compile-time and run-time [4, 9]. Developers use security APIs to manifest security policies tailored to their applications. Together, protection models and security APIs comprise the state of the art for safeguarding applications running in open 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 systematically embedding security functionality within applications [12]. Developers use special syntax to express security policies within code in the same way that types are used to ex-

press 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 occur within a program.

This paper introduces a methodology for extending programming languages with programmable security services. The methodology is first described, followed by the authorization model adopted for programmable access control. Finally, the design and prototype implementation of our programmable access control solution in Java is presented.

2. PROGRAMMABLE SECURITY

Programmable security links security services to programming language syntax extensions to facilitate the expression and systematic implementation of security policies in applications. Developers use programmable security expressions to specify authorization policies for principals and data elements, authentication protocols for proxies, audit procedures for program modules, and secure communication channels for distributed systems.

The implementation of native programmable security services in new languages offers language architects greater freedom, allowing them to escape the "golden handcuffs" of compatibility. However, extending popular languages has the advantage of immediate relevance to a large audience of developers.

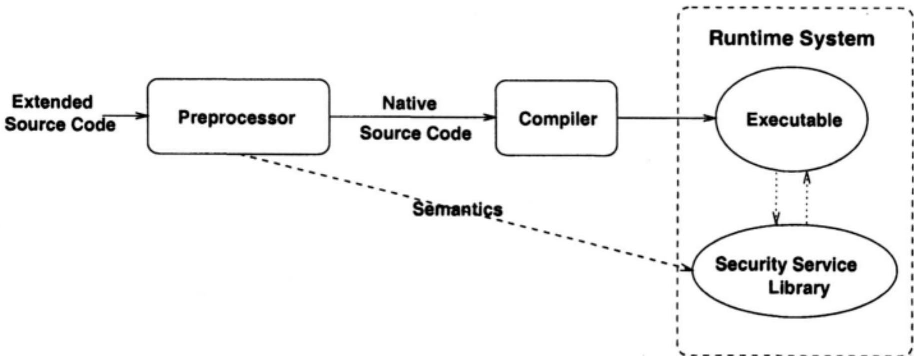

Figure 1. Programmable security methodology.

Figure 1 illustrates an extensional methodology for implementing programmable security. This approach introduces two elements to a programming system; a preprocessor (a similar preprocessing approach has been taken to add genericity to the Java language [3]) and a security service library. The preprocessor employs a parser designed with an augmented grammar to accept expressions in the extended language.

The augmented grammar adds production rules and keywords to the original grammar, binding security services to the programming system. The preprocessor actually plays two roles; (i) checking for security violations within the extended source code and (ii) translating the extended source code into native source code.

The preprocessor relies on the security service library API to derive native source code implementing the specified security functionality. The resulting source code is passed to a native compiler to produce an executable. The security service library can be linked at compile-time or run-time to perform security violation checks during execution.

3. AUTHORIZATION MODEL

Our programmable access control solution relies on a simplified version of the ticket-based authorization model originally described in [12, 13, 14] and refined in [6]. We adopt the ticket-based scheme because it permits policy expression in a variety of authorization models and a straightforward implementation in message passing systems.

Messages requesting access to remote resources are between nodes in an object hierarchy (where any object can also play the role of a subject). Tickets embedded in messages as unforgeable tokens are analogous to capabilities [7, 8, 16, 17], conveying privileges of message originators. Message passing only occurs directly between two adjacent object nodes, as formally specified with the following rule:

$$(1) \quad \mathit{Adj}\ s\ o_1\ o_2 = \mathit{Parent}\ s\ o_1\ o_2 \lor \mathit{Parent}\ s\ o_2\ o_1.$$

Adj s o_1 o_2 is true whenever objects o_1 and o_2 are adjacent in a hierarchical object structure at a given state s. *Parent* s o_1 o_2 is true when, in state s, o_1 is the parent of o_2.

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

Key s o_1 t is true when o_1 has a key named t in state s. *Lock* s o_1 o_2 t is true when o_1 has a lock named t on object o_2 in state s. (The hierarchy described below mandates that o_1 and o_2 be *adjacent* for o_1 to hold such a lock.) We can define key/lock matching by an object o_2 as

$$(2) \quad \mathit{Match}\ s\ o_1\ o_2\ o_3 = \exists\, t.\ \mathit{Key}\ s\ o_1\ t \land \mathit{Lock}\ s\ o_2\ o_3\ t.$$

This predicate defines when a message has permission to access o_3 on behalf of o_1 in o_2. Every message must be authorized for delegation at every inteverning object in the hierarchy. The access request itself is checked only at the destination object.

Another predicate represents the 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 a hierarchy with the addition of two rules:

$$(3)\qquad Access\ s\ o_1\ o_1\ =\ true$$

and

$$(4)\qquad Access\ s\ o_1\ o_2\ \wedge\ Match\ s\ o_1\ o_2\ o_3\ \Rightarrow\ Access\ s\ o_1\ o_3.$$

Rule 3 indicates that objects always have access to themselves, and forms a base case for inductive access checking. Rule 4 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.

4. PACKAGE-BASED ACCESS CONTROL

This section presents a programmable package-based protection scheme for Java. The system (JPAC) uses 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. Developers can use JPAC to confine access to program elements based on the package identity of requesting elements.

Figure 2 presents the JPAC syntax extensions used to express package-based protection. Extensions are based on the syntax described in The Java Language Specification documents [2, 21, 22, 23]. The EBNF productions in Figure 2 change the way a compilation unit is named by making `PackageDeclaration` mandatory and adding a `Properties` production to it. Unnamed compilation units are specified by declaring a nameless package.

Three examples of legal compilation units can be found at the bottom of Figure 2. A package `faculty` specifies that its elements can be accessed by a package named `admin`. Associating the keyword `guarded` with the `student` package specifies that its elements can be accessed by any other package using our ticket-based protection scheme. The package `other`, using the keyword `unsecured`, specifies that any package can access its elements, even those that are not compiled under our architecture (useful for interfacing with APIs or other external code).

Synchronization clauses or exceptions are not controlled in our design. Public, private, protected and package-level protection modes are enforced as usual, with package-based protection specifications imparting additional authorization constraints.

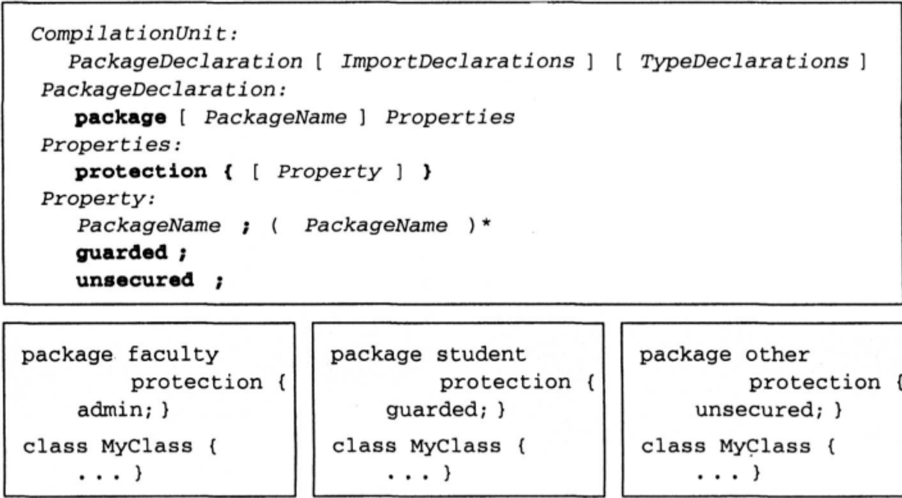

Figure 2. Extended syntax.

The JPAC semantics are derived from the ticket-based authorization scheme and object hierarchy described earlier. Every JPAC system consists of a root object, and within it a collection of objects mapped to JPAC-protected packages – each associated with its own message handler. Classes and instances are regarded as components of their enclosing packages and use their package's message handler to pass messages conveying access requests/replies across package boundaries.

A unique token is defined for each protected package. A lock/key pair is generated from the token for each of the packages listed in the protection declaration clause. The lock is held in the protected package's access control list, while keys are delivered to the packages for whom access is granted. A special token representing "everyone" is defined to build a key for distribution to all objects in the JPAC system. Packages with guarded protection status hold the "everyone" lock, enabling access by all JPAC objects, but not by external Java code.

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. 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.

Our prototype implements package protection with filter operations performed by message handlers. Messages contain fields identifying the source, the destination and the keys of their originator. Each time a

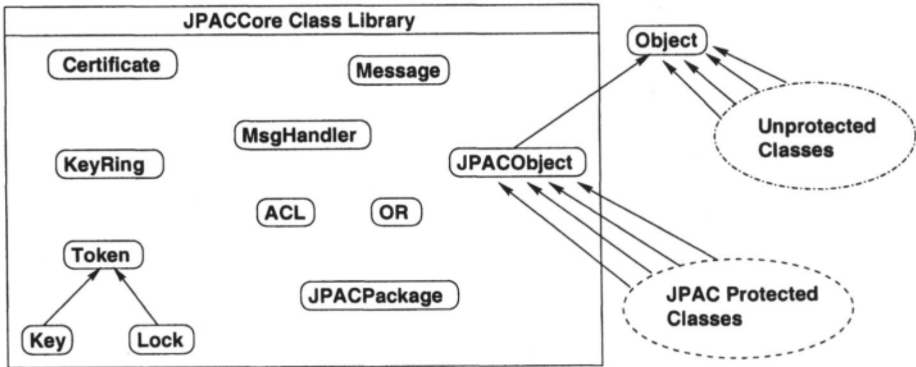

Figure 3. JPAC class hierarchy.

message is received by a message handler it verifies keys contained in the message against locks held in a package access control list.

5. JPAC IMPLEMENTATION

JPAC integrates a set of classes to model a message passing hierarchy for program elements, a preprocessor for extended source code translation, and a run-time ticket management software architecture. The class hierarchy comprising the security service library is shown in Figure 3.

5.1. OBJECT HIERARCHY

The JPAC implementation adds a `MessageHandler` class and a class for each package to dispatch access control in an object hierarchy. Instances of `MessageHandler` are associated with each of the newly created package classes. Package classes hold package-specific data, guiding `MessageHandler` initialization. All package classes inherit their behavior from the `JPACPackage` abstract class.

Figure 4 shows the source code for a JPAC class `Learn`, including a protection clause restricting access to the `faculty` package. Resulting package classes are named as "Package_X", where "X" is a flattened version of the package name. This simple naming convention for package classes helps avoid name clashes in JPAC.

5.2. PREPROCESSING

The preprocessor performs various transformations on program elements in extended source code to authenticate calling subjects, effect secure method dispatch and respect all forms of variable access. Variable and method access is validated with a certificate placed in an extra

Learn.jpac:

```
package student protection {
faculty;
}

public class Learn extends tools.BBSClient {
    public int addMsg(String msg) {
        super.addMsg(msg);
        return noOfMsgs++;
    }
    protected int version = 1.0;
    // the method public int getNoOfMsgs() and the field
    // public int noOfMsgs are inherited from class tools.BBSClient
}
```

Figure 4. Package-based Java protection code.

parameter generated by the preprocessor. Finally, the preprocessor must successfully integrate Java interfaces and unsecured packages with JPAC systems.

5.2.1 Methods. JPAC method calls are transformed to include an authenticating certificate placed in the extra parameter generated by the preprocessor. The method `checkOut()` in the `MessageHandler` of the current package class checks if a message can reach the destination, and returns a certificate for the callee method.

Extending Java's protection model to permit discretionary access control provides a unique set of challenges to the preprocessor. For example, protected classes may invoke unsafe methods in inherited classes. Protected classes in JPAC inherit from the `JPACObject` class, which provides safe replacements for methods in `java.lang. Object.` If a method from Object is used, a `JPACAccesDeniedException` exception is thrown.

Inheritance and dynamic linking in Java also produce an interesting problem. Any call destined for a class in some package can arrive, due to inheritance, to another class in a different package. If this is allowed, legitimate calls could be denied just because inheritance was used. The JPAC solution is to produce proxy methods in the inheriting class that forward calls to the parent class.

Another complication results from the fact that all constructors call their parent's default constructor. When a call to the parent's constructor is missing in a JPAC constructor, one is placed with a fresh certificate in the first statement of the constructor body.

5.2.2 Variables. Direct variable reads and writes are emulated with accessor methods, which authorize access by validating certificates in the same way as JPAC methods. These accessor methods inherit protection levels from their variables. Furthermore, if a variable is static

then the accessor method is made static as well. Variables in the JPAC code are then set to private-level protection in the transformed Java code, preventing unauthorized access.

JPAC accessor method names are chosen to reflect not only the name of the variable, but also the class (or interface) that contains it. When variable accesses are found in JPAC code, the preprocessor determines the type where the variable is stored and generates the appropriate call as needed.

5.2.3 Interfaces and Unprotected Packages. JPAC interfaces are translated similar to classes, except that interface variables are implicitly static, public and final, making them impossible to "privatize." JPAC moves interface variables into specially constructed classes making it possible to change their access level to private and to add accessor methods. Wherever a JPAC interface variable is referenced, a call to the appropriate accessor method is substituted. Naming of interface variables and accessor methods is performed in the same way as for class variables.

Classes and interfaces in unsecured packages are considered unprotected by the JPAC system extensions. Unsecured classes and interfaces are useful in that they can directly subclass `Object` (the core Java system class) and other unprotected classes. Classes and instances in unsecured packages do not adopt the additional certificate parameter for methods or accessor methods for variables. The only modifications are the transformation of calls to methods and variables in classes belonging to protected packages and the addition of a package class.

5.3. TICKET MANAGEMENT

The ticket management software architecture in Figure 5 serves as an execution model for the ticket-based access control scheme described earlier. The architecture supports message passing in a hierarchy of handlers to validate inter-package access.

5.3.1 Keys, Locks and Rings. Key and lock objects provide a basis for constraining access, while messages encapsulate authorization requests. Tokens, which model keys and locks, are characterized by an abstract class (`Token`) containing a unique identifier and an abstract method to match keys and locks. A simple protocol ensures that a lock and its matching key are created simultaneously to guarantee their authenticity.

The `match()` method checks keys and locks by computing an encrypted key value and comparing it to the value of the lock.

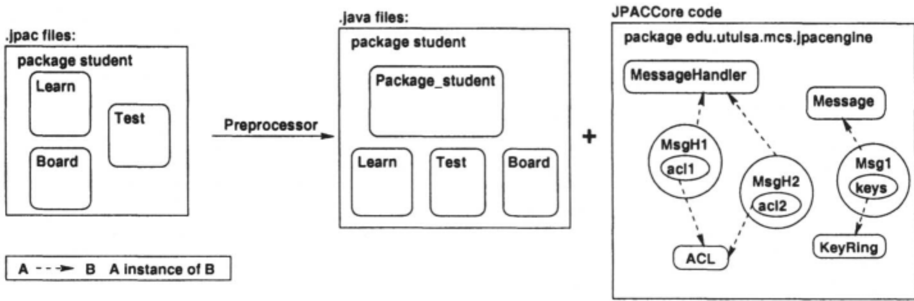

Figure 5. Software architecture.

Access control lists maintained within each package class store locks and keys. The access control list class (`ACL`) provides methods to add and remove keys from a key ring. Instances of the `KeyRing` class are passed between message handlers to facilitate inter-package access by presenting keys to match with locks in destination package access control lists.

To access a variable or method in another package, the key ring is passed as a parameter to the `match()` method of the destination `ACL`. The `match()` method passes each lock to a method of the key ring which checks every key for a match. If a key fits, the key ring returns a copy of the key to the `ACL` for verification, and access is granted. When a foreign package attempts to gain access to a local package, the local package receives a copy of the foreign key registry.

5.3.2 Message Handlers. The centerpiece of the software architecture is the message handler, which effects message passing between objects. Messages are used only for authorization and consist of a key ring, a source and destination identifier, and a certificate. Key rings that accompany messages embody the rights of the message originator. Messages are passed between objects via resident message handlers, authorized at each stop, until they reach their final destination.

Static methods in the `MessageHandler` model behavior for the root JPAC domain. A static code block creates a new message handler for each package class. The `MessageHandler` constructor obtains a reference to program units above it in the hierarchy. If such a reference is to a unit that has not yet been instantiated, that unit's static code will execute, creating its parent object. The result of this domino effect is that when any package is first accessed, message handlers for it and every other package above it in the hierarchy are initialized.

6. COMPARISONS WITH OTHER WORK

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 [2, 4]. Java 1.2 lets developers define protection domains to specify sets of classes sharing identical permissions [10, 11]. 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 [26]. 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 [1] provides a flow logic to verify that programs satisfy confinement properties. Static analysis of security properties has re-emerged as a promising line of research because it eliminates much of the need for costly runtime checks. It also prevents information leakage that can occur at runtime.

In [25], 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 to support the static analysis of secure program interoperability.

Van Doorn *et al.*, extend Modula-3 network objects with security features in [24]. Secure network objects (SNOs) bind programming languages into service for integrating security into objects and methods. SNOs promote subtyping for specifying security properties of objects.

Myers and Liskov describe a decentralized information flow control model in [19]. Security label annotations can be inferred and type-checked by a special compiler to verify program information flow properties. Myers implemented these ideas in JFlow, a variant of Java that integrates statically checked information flow annotations with advanced programming language features such as objects, subclassing and exceptions [18].

The SLam calculus is a typed **λ-calculus** that tracks relevant security information of programming elements [15]. A compiler that executes static checks enforces the type system rules to guarantee program security. Types in SLam are monomorphic and static, but the system has been shown to be extensible to concurrent and imperative programming.

7. CONCLUSIONS

Programmable security allows developers to express verifiable protection policies with special syntax. Preprocessors can be used to extend existing programming languages with syntactic constructs tied to security service libraries, yielding a programmable solution that is interoperable with systems developed in the original language. Our programmable access control prototype, JPAC, extends the Java programming 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. In addition, the semantic foundation of the JPAC architecture permits the design and implementation of more fine-grained authorization models for class-based and instance-based protection.

References

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

[2] Arnold, K. and Gosling, J. (1998) *The Java Programming Language*, *2nd Edition*. Addison-Wesley, Reading, Massachusetts.

[3] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P. (1998) Making the future safe for the past: Adding genericity to the Java programming language *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)* ACM SIGPLAN Notices **33(10)**, 183–200.

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

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

[6] Dionysiou, I. (2000) *A Formal Semantics for Programmable Access Control*, Masters Thesis, Washington State University.

[7] Fabry, R. (1974) Capability-based addressing. *Communications of the ACM*, **17(7)**, 403–412.

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

[9] Gong, L. (1998) Secure Java class loading. *IEEE Internet Computing*, **2(6)**, 56–61.

[10] Gong, L., Mueller, M., Prafullchandra, H. and and Schemers, R. (1997) 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*, 103–112.

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

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

[13] Hale, J., Threet, J. and Shenoi, S. (1998) 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, 134–150.

[14] Hale, J., Threet, J. and Shenoi, S. (1997) A framework for high assurance security of distributed objects, in *Database Security, X: Status and Prospects* (eds. P. Samarati and R. Sandhu), Chapman and Hall, London, 101–119.

[15] Heintze, N. and Riecke, J. (1998) The SLam calculus: Programming with security and integrity. *Proceedings of the Twenty-Fifth ACM SIGPLAN-SIGACT on Principles of Programming Languages*, 365–377.

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

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

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

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

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

[21] Sun Microsystems. (1999) Clarifications and Amendments to The Java Language Specification, http://www.java.sun.com/docs/books/jls/clarify.html.

[22] Sun Microsystems. (1997) Inner Classes Specification. http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html.

[23] Sun Microsystems. (1999) Clarifications and Amendments to the Inner Classes Specification, http://www.java.sun.com/docs/books/jls/nested-class-clarify.html.

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

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

[26] Wallach, D., Balfanz, D., Dean, D. and Felten, E. (1997) Extensible security architectures for Java. *Proceedings of the 16th Symposium on Operating Systems Principles*, 116–128.