SECURE JAVA DEVELOPMENT WITH UML

Jan Jürjens*
Computing Laboratory, University of Oxford, GB
http://www.jurjens.de/jan – jan@comlab.ox.ac.uk

This is version 21/11/01 of a paper at INetSec'01. Please refer to www.jurjens.de/jan for the current version and more information.

Abstract Developing secure software systems is difficult and error-prone. Numerous implementations have been found vulnerable in the past; a recent example is the unauthorised access to millions of online account details at an American bank.

We aim to address this general problem in the context of development in Java. While the JDK 1.2 security architecture offers features (such as guarded objects) that provide a high degree of flexibility and the possibility to perform fine-grained access control, these features are not so easy to use correctly.

We show how to use a formal core of the Unified Modeling Language (UML), the de-facto industry-standard in object-oriented modelling, to correctly employ Java security concepts as such as signing, sealing, and guarding objects. We prove results for verification of specifications wrt. security requirements. We illustrate our approach with a (simplified) account of the development of a web-based financial application from formal specifications.

Keywords: Distributed systems security, access control, mobile code, Java security, secure software engineering, Unified Modeling Language.

1. Introduction

The need to consider security aspects in the development of many systems today is not always met by adequate knowledge on the side of the developer. This is problematic since in practice, security is compromised most often not by breaking the dedicated mechanisms (such as encryption or access control), but by exploiting weaknesses in the way they are being used [And01]. Thus security mechanisms cannot be

*Supported by the Studienstiftung des deutschen Volkes and the Computing Laboratory.

"blindly" inserted into a security-critical system, but the overall system development must take security aspects into account.

Especially dynamic access control mechanisms such as provided by Java since the JDK 1.2 security architecture [Gon99; Kar00b] in the form of GuardedObjects can be difficult to administer since it is easy to forget an access check [Gon98; BV99]. If the appropriate access controls are not performed, the security of the entire system may be compromised. Additionally, access control may be granted indirectly and unintentionally by granting access to an object containing the signature key that enables access to another object. In this work, we aim to address these problems by providing means of reasoning about the correct deployment of security mechanisms such as *signed*, *sealed* and *guarded objects* using a formal core of the widely used object-oriented design language Unified Modeling Language (UML), extending previous work [Jür01f; Jür01a].

The more general aim of this work is to use UML to encapsulate knowledge on prudent security engineering and thereby make it available to developers not specialised in security [Jür01b]. Thus the approach to use UML for security covers not just access control, but also other security functions and requirements.

Overview. After presenting some background on access control in Java in the following section, we summarise our use of UML in section 3. In Section 4 we outline the part of a design process relevant to enforcing access control in Java and give some results on verifying access control requirements. In Section 5 we illustrate our approach with the example of the development of a web-based financial application from formal specifications. We end with an account of related work, a conclusion and indication of future work. Proofs have to be omitted due to space reasons and will appear in an extended version.

2. Access control in Java

Authorisation or access control [SS94] is one of the corner-stones of computer security. The objective is to determine whether the source of a request is *authorised* to be granted the request. Distributed systems offer additional challenges: The trusted computing bases (TCBs) may be in various locations and under different controls. Communication is in presence of possible adversaries. Mobile code is employed that is possibly malicious. Further complications arise from the need for delegation (i. e. entities acting on behalf of other entities) and the fact that many security requirements are location-dependent (e.g., a user may have more rights at the office terminal than when logging in from home).

Object-oriented systems offer a very suitable framework for considering security due to their encapsulation and modularisation principles [FDR94; Var95; ND97; Gol99; Sam00].

In the JDK 1.0 security architecture, the challenges posed by mobile code were addressed by letting code from remote locations execute within a *sandbox* offering strong limitations on its execution. However, this model turned out to be too simplistic and restrictive. From JDK 1.2, a more fine-grained security architecture is employed which offers a user-definable access control, and the sophisticated concepts of signing, sealing, and guarding objects [Gon99; Kar00b].

A protection domain [SS75] is a set of entities accessible by a principal. In the JDK 1.2, permissions are granted to protection domains (which consist of classes and objects). Each object or class belongs to exactly one domain.

The system security policy set by the user (or a system administrator) is represented by a policy object instantiated from the class java.security.Policy. The security policy maps sets of running code (*protection domains*) to sets of access permissions given to the code. It is specified depending on the origin of the code (as given by a URL) and on the set of public keys corresponding to the private keys with which the code is signed.

There is a hierarchy of typed and parameterised access permissions, of which the root class is java.security.Permission and other permissions are subclassed either from the root class or one of its subclasses. Permissions consist of a target and an action. For file access permissions in the class FilePermission, the targets can be directories or files, and the actions include read, write, execute, and delete.

An access permission is granted if all callers in the current thread history belong to domains that have been granted the said permission. The history of a thread includes all classes on the current stack and also transitively inherits all classes in its parent thread when the current thread is created. This mechanism can be temporarily overridden using the static method doPrivileged().

Also, access modifiers protect sensitive fields of the JVM: For example, system classes cannot be replaced by subtyping since they are declared with access modifier final.

The sophisticated JDK 1.2 access control mechanisms are not so easy to use. The granting of permissions depends on the execution context (which however is overridden by doPrivileged(), which creates other subtleties). Sometimes, access control decisions rely on multiple threads. A thread may involve several protection domains. Thus it is not always easy to see if a given class will be granted a certain permission.

This complexity is increased by the new and rather powerful concepts of signed, sealed and guarded objects [Gon99]. A SignedObject contains the (to-be-)signed object and its signature.[1] It can be used internally as an authorisation token or to sign and serialise data or objects for storage outside the Java runtime. Nested SignedObjects can be used to construct sequences of signatures (similar to certificate chains).

Similarly, a SealedObject is an encrypted object ensuring confidentiality.

If the supplier of a resource is not in the same thread as the consumer, and the consumer thread cannot provide the access control context information, one can use a GuardedObject to protect access to the resource. The supplier of the resource creates an object representing the resource and a GuardedObject containing the resource object, and then hands the GuardedObject to the consumer. A specified Guard object incorporates checks that need to be met so that the resource object can be obtained. For this, the Guard interface contains the method checkGuard, taking an Object argument and performing the checks. To grant access the Guard objects simply returns, to deny access is throws a SecurityException. GuardedObjects are a quite powerful access control mechanism. However, their use can be difficult to administer [Gon98]. For example, guard objects may check the signature on a class file.

3. Developing Secure Systems with UML

To address these issues, we extend previous work [Jür01f; Jür01a] to employ a formal core of the Unified Modeling Language (UML) [UML01], the de-facto industry standard in object-oriented modelling (an excellent introduction is given in [SP00]). We would like to ensure that the protection mechanisms that are in place do offer the required level of security. Specifically, we check the specified dynamic behaviour against expressed security policies. We do this on the level of specification (rather than the implementation level) because design mistakes can so be corrected as early as possible, and because formal reasoning is more feasible at a more abstract level.

UML consists of several kinds of diagrams describing the different views on a system. We use only a simplified fragment of UML (together with a formal semantics) to enable formal reasoning and keep the presentation concise. We use its standard extension mechanisms to express security aspects. As a formal semantics for UML is subject of ongoing research, we use a (simplified) semantics tailored to our needs for the

[1] Note that signing object is different from the signing of JAR files.

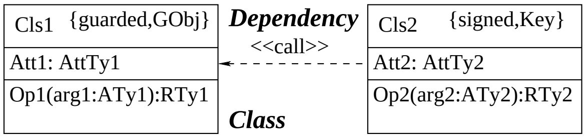

Figure 1. Class diagram

time being, just to illustrate our ideas. Note, however, that our approach does not rely on use of a formal semantics; in fact we aim for a tool to automatically check the considered security notions, and then these may also be explained informally (which is more accessible, but may be more prone to misunderstanding).

We use the following kinds of diagrams: class diagrams, statechart diagrams, and deployment diagrams.

We define the diagrams using their abstract syntax for conciseness and to enable formal reasoning. We also give the concrete syntax (in a way that the translation between the two should be apparent).

3.1. Class Diagrams

Using class diagrams we can model which objects are signed or sealed with which keys, and which are guarded by which Guard objects.

An *attribute specification* $A = (\mathsf{att\_name}, \mathsf{att\_type}, \mathsf{init\_value})$ is given by a name att_name, a type att_type and an initial value init_value.

An *operation specification* $O = (\mathsf{op\_name}, \mathsf{Arguments}, \mathsf{op\_type})$ is given by a name op_name, a set of Arguments and the type op_type of the return value. The set of arguments may be empty and the return type may be the empty type $\emptyset$ denoting absence of a return value. An *argument* $A = (\mathsf{arg\_name}, \mathsf{arg\_type})$ is given by its name arg_name and its type arg_type.

A *class model* $C = (\mathsf{class\_name}, (\mathsf{tag}, \mathsf{value}), \mathsf{AttSpecs}, \mathsf{OpSpecs}, \mathsf{State})$ is given by a name class_name, an optional $(\mathsf{tag}, \mathsf{value})$ pair (written in curly brackets), a set of attribute specifications AttSpecs, a set of operation specifications OpSpecs and a statechart diagram State giving the object behaviour. The tag may be either of signed, sealed or guarded (indicating a signed, sealed or guarded object), and the value is either the public key corresponding to the private key with which the object was signed or sealed, or it is the name of the corresponding Guard object.

A *class diagram* $D = (\mathsf{Cls}, \mathsf{Dependencies})$ is given by a set Cls of class models and a set of Dependencies. A *dependency* is a tuple (client, supplier, stereotype) consisting of class names client and supplier and a label (called stereotype) indicating the kind of dependency (e.g. «*call*»).

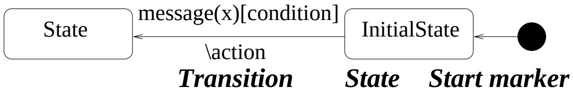

Figure 2. Statechart Diagrams

3.2. Statechart diagrams

We use statechart diagrams to specify the behaviour of objects, in particular of the Guards.

We fix a set Var of (typed) variables $x, z, y, \ldots$. We define the notion of a statechart diagram for a given class model C: A *statechart diagram* $S = (\mathsf{States}, \mathsf{init\_state}, \mathsf{Transitions})$ is given by a set of States (that includes the initial state init_state) and a set of Transitions. (In the concrete syntax, the initial state is signified with a start marker.)

A *statechart transition* $t = (\mathsf{source}, \mathsf{event}, \mathsf{condition}, \mathsf{Actions}, \mathsf{target})$ has a source state, an event, a condition, a list of Actions and a target state. An *event* is the name of an operation with a list of distinct variables as arguments (e.g. $\mathsf{op}(x, y, z)$). Let the set Assignments consist of all partial functions that assign to each variable and each attribute of the class C a value of its type. A *condition*[2] is a function $g : \mathsf{Assigments} \to \mathsf{Bool}$ evaluating each assignment to a boolean value. We write it as a sequence of Boolean propositions with variables and attribute names that is interpreted as their conjunction; conditions are written in square brackets. An *action* can be either to assign a value v to an attribute a (written $a := v$), to call an operation op with values $v_1, \ldots, v_n$ (written $\mathsf{op}(v_1, \ldots, v_n)$), to return values $v_1, \ldots, v_n$ as a response to an earlier call of the operation op (written $\mathsf{return}_{\mathsf{op}}(v_1, \ldots, v_n)$), or to throw an exception. In each case, the values can be constants, variables or attributes. In the concrete syntax, actions are preceded by a backslash.

3.3. Deployment diagrams

Deployment diagrams describing the physical layer of a system are security-relevant in so far as they give the locations of the different components of the system (used in the access permissions) and they give information on kinds of the communication links between different components, inducing threat scenarios wrt. the physical security.

A system *node* $N = (\mathsf{location}, \mathsf{Components})$ is given by its location (e.g. a URL or “local system”) and a set of contained Components.

[2] We do not use the UML term *guard* here to avoid confusion with guard objects.

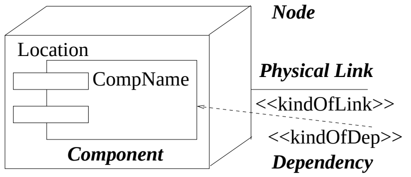

Figure 3. Deployment diagram

A *deployment diagram* $D = (\mathsf{Nodes}, \mathsf{Links}, \mathsf{Dependencies})$ is given by a set of Nodes, a set of communication Links between nodes and a set of logical Dependencies between components. A *link* $l = (\mathsf{nds}, \mathsf{stereo})$ consists of a two-element set nds of nodes being linked and a label (called *stereotype*) indicating the kind of the link (e.g. «*Internet*»). Here a *dependency* is a tuple (client, supplier, tag) consisting of components client and supplier and a label (called tag) indicating the kind of dependency (e.g. {rmi}).

4. Design process

We sketch the part of a design process for secure systems using UML that is concerned with access control enforcement using guarded objects.

(1) Formulate the permission sets for access control for sensitive objects.

(2) Use statecharts to specify Guard objects that enforce appropriate access control checks.

(3) Verify that the Guard objects protect the sensitive objects sufficiently by showing that they only grant access implied by the security requirements.

(4) Ensure that the access control mechanisms are consistent with the functionality required by the system by showing that the other objects may perform their intended behaviour.

(5) Verify that mobile objects are sufficiently protected by considering the threat scenario arising from the physical layer given in the deployment diagram.

Here the access control requirements in step (1) can be of the following form:[3]

[3] In future work we intend to formalise these requirements using an abstract security policy specification language, enabling automatic generation of the corresponding guard object specifications.

- origin of requesting object (based on URL)
- signatures of requesting object
- external variables (such as time of day etc.).

In Section 5 we sketch a formal verification of a specification following these steps. They enforce the following two requirements.

Security requirement: Check that the access control requirements are strong enough to prevent unauthorised influence, given the threat scenario arising from the physical layer.

Functionality requirement: Check that the access control requirements formulated are not overly restrictive, denying legitimate access from other components of the specification.

The functionality requirement is important since it is not always easy to see if stated security requirements are at all implementable. If their inconsistency is only noticed during implementation then, firstly, resources are wasted since work has to be redone. Secondly, most likely security will be degraded in order to reduce this extra work.

4.1. Verification

In this subsection, we sketch results to be applied in the above approach. The idea is to verify security properties by linking the different views on a system given by the various kinds of diagrams. We convey our ideas using a simplified semantics for UML statechart diagrams.

Any statechart diagram S defines a function $[\![S]\!]$ from sequences of input events to sets of sequences of output actions, each possibly with arguments, often involving use of cryptographic operations (as detailed in [Jür01f]). We say that S *may eventually output* a value v if there exists a sequence $\vec{e}$ of input events and a sequence $\vec{a} \in [\![S]\!](\vec{e})$ of corresponding output actions such that v is output by one of the actions in $\vec{a}$ (in cleartext) [Jür01e].

The following definition uses the notion of an *adversary* from [Jür01e], which is a function from sequences of output actions of the statechart S to sequences of input events of S that captures the capabilities of an adversary intercepting the «*Internet*» communication links between S and the other objects (the exact definition of “adversary”, “without prior knowledge” and of the composition $\otimes$ of the statechart interpretation $[\![S]\!]$ with the adversary A can be found in [Jür01e]).

Definition 1 A statechart diagram S *preserves the secrecy* of a value K if there is no adversary A (eavesdropping on the «*Internet*» links) without prior knowledge of K such that $[\![S]\!] \otimes A$ may eventually output K.

This definition is extended to system components by composing the functions arising from the statechart diagrams specifying the objects of a given component.

Intuitively, then, a system component C preserves the secrecy of K if no adversary can find out K in interaction with the system modeled by C, following the approach of Dolev and Yao (1983), cf. [Aba00; Jür01e].

The following result is applied within the approach of subsection 4 to the UML specification of a security-critical system (for a proof of this as well as the following results cf. [Jür01d]).

Theorem 1 *Suppose that the access to a certain resource is according to the* Guard *object specifications granted only to objects signed with a key K. Suppose all components preserve the secrecy of K. Then only objects signed with K according to the specification will be granted access to the resource.*

5. Example Financial Application

We illustrate our approach with the example of a web-based financial application. The example was chosen to be tractable enough given the space restrictions but still realistic in that it points out some typical issues when considering access control for web-based e-commerce applications (namely to have several entities – service-providers and customers – interacting with each other while granting the other parties a limited amount of trust and by enforcing this using credentials).

We first describe the physical layer of the application in a UML diagram and state its security requirements. We show in UML diagrams how to employ GuardedObjects to enforce these security requirements. We prove that the specification given by the UML diagrams is secure by showing that it does not grant any access not implied by the security requirements. We end the section by giving supplementary results regarding consistency of the security requirements.

Two (fictional) institutions offer services over the Internet to local users: an Internet bank, Bankeasy, and a financial advisor, Finance. The physical layer is thus given in Figure 4.

To make use of these services, a local client needs to grant the applets from the respective sites certain privileges.

(1) Applets that originate at and are signed by the bank can read and write the financial data stored in the local database, but only

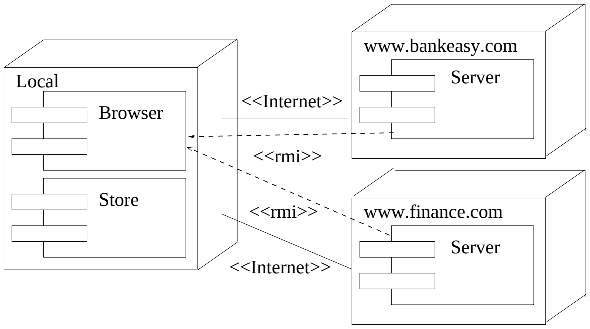

Figure 4. Deployment diagram

between 1 pm and 2 pm (when the user usually manages her bank account).

(2) Applets from (and signed by) the financial advisor may read an excerpt of the local financial data created for this purpose. Since this information should only be used locally, they additionally have to be signed[4] by a certification company, CertiFlow, certifying that they do not leak out information via covert channels.

(3) Applets originating at and signed by the financial advisor may use the micropayment signature key of the local user (to purchase stock rate information on behalf of the user), but this access should only be granted five times a week.

Financial data sent over the Internet is signed and sealed to ensure integrity and confidentiality. Access to the local financial data is realised using GuardedObjects. Thus the relevant part of the class diagram is given in Figure 5.

As specified in the class diagram, the access controls are realised by the Guard objects FinGd, ExpGd and MicGd, whose behaviour is specified in Figures 6, 7 and 8 (we assume that the condition timeslot is fulfilled if and only if the time is between 1pm and 2pm, that the condition weeklimit is fulfilled if and only if the access to the micropayment key has been granted less than five times in the current calendar week, and that the method incThisWeek increments the relevant counter).

Now according to step (3) in Section 4, we prove that the specification given by UML diagrams is secure in the following sense.

[4] Here we assume that SignedObject is subclassed to allow multiple signatures on the same object [Gon99].

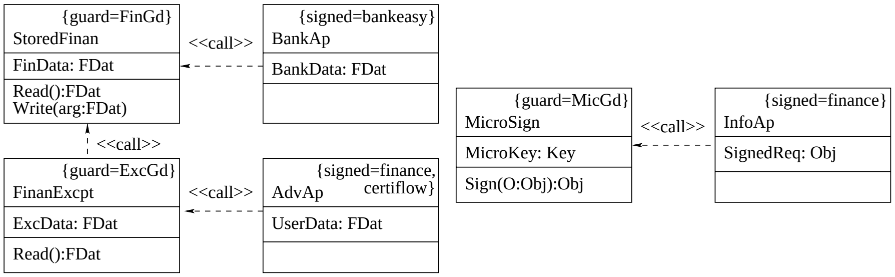

Figure 5. Class diagram

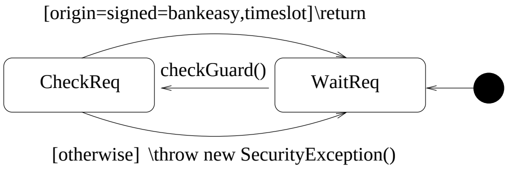
[origin=signed=bankeasy,timeslot]\return
CheckReq
checkGuard()
WaitReq
[otherwise] \throw new SecurityException()

Figure 6. Statechart FinGd

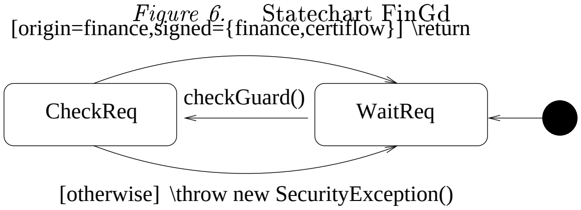
[origin=finance,signed={finance,certiflow}] \return
CheckReq
checkGuard()
WaitReq
[otherwise] \throw new SecurityException()

Figure 7. Statechart ExcGd

Theorem 2 *The specification given by UML diagrams for the guard objects does not grant any permissions not implied by the access permission requirements given in (1)–(3).*

Regarding step (4) in Section 4, we exemplarily prove that InfoAp can purchase the article on behalf of the user, as intended.

Theorem 3 *Suppose all applets in the current execution context originate from and are signed by Finance, and that use of the micropayment key is requested, which has happened less than five times before in the current week. Then the current applet is permitted to purchase articles on behalf of the user.*

Finally, following (5) in Section 4, the mobile objects are sufficiently protected since all objects sent over the Internet were required to be signed and sealed (a more detailed discussion has to be omitted).

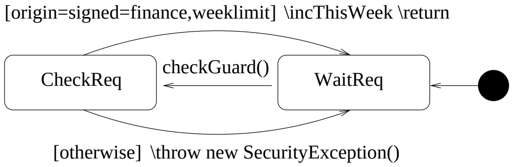
[origin=signed=finance,weeklimit] \incThisWeek \return
CheckReq
checkGuard()
WaitReq
[otherwise] \throw new SecurityException()

Figure 8. Statechart MicGd

6. Related Work

In [Jür01f; Jür01a] we considered how to model various aspects of general systems security (including multi-level security, secure information flow and security protocols) with UML. [Jür01c] applies UML to reason about audit-security in a smart-card based payment scheme and [Jür01b] shows how to use UML to enforce general principles of secure systems design from [SS75]. There seems to be little other systematic work yet in applying UML to security.

Java 2 security and in particular the advanced topics of signed, sealed and guarded objects is explained in [Gon99]. There has also been some work giving formal reference models for Java 2 access control mechanisms, thus clarifying possible ambiguities in the informal accounts and enabling proof of compiler conformance to the specification [KG98; WF98; Kar00b] (but without considering signed, sealed or guarded objects). To our knowledge, the use of signed, sealed or guarded objects in JDK 1.2 has not previously been considered in a formal model.

[HKK00] introduces higher-level abstractions for Java security policy rules, simplifies security management and gives additional functionality. General Java security is considered e.g. in [GAS99].

There has been extensive work regarding formal models for security, mostly about security protocols (for an overview cf. [GSG99; RSG$^+$01]). A logic for access control was introduced in [ABLP93].

7. Conclusion and Future Work

To summarise, we used a core of UML, the industry standard in object-oriented modelling, to specify and reason about access control in distributed Java-based systems. We have concentrated on advanced JDK 1.2 access control mechanisms such as signing, sealing and guarding objects. We show how to specify security requirements and to prove that modelled access control mechanisms such as guarded objects meet their goals and that these mechanisms are consistent with the overall functionality required from the system.

In conclusion, it seems that our approach is both worthwhile and feasible:

- Using the JDK 1.2 access control mechanisms can be rather complicated in practice (especially when indirect access permissions using authorisation tokens are employed), thus providing support for correct specification of the relevant mechanisms in the context of a widely used specification as UML seems quite useful.
- In this paper, we could only illustrate our approach using a rather simple example. However, UML allows a high degree of abstraction in modelling systems. So we expect the approach to scale up rather well. This is currently validated in practice in a Master's thesis developing an Internet-based auction system [Mea01].

A further benefit is that by using a widely accepted notation, our approach to secure Java development can be integrated with other work on secure systems using UML (e.g. on electronic purse systems [Jür01c]).

As to the limitations of this first step in this direction of research, our account remains relatively abstract for space restrictions and conciseness of presentation. As a next step, one should consider more details of Java security, such as the use of access modifiers (private, final,...), the doPrivileged() method and the implies() method. Also, an extension to JAAS [LGK^{+}99; Kar00b] is planned.

Work in progress aims to provide tool support to validate UML specifications of access control guards against security requirements, building on work in [CCR01].

Regarding future work, it would be very useful to have a way to generate the correct behaviour specification of guard objects in statechart diagrams automatically from the (formalised) security requirements. Also, it would be interesting to try to extend our approach to the extension of the Java security architecture proposed in [HKK00]. We intend to address CORBA security (cf. e.g. [VH96; Kar00a]) in a similar way.

Acknowledgments

The idea for the line of work using UML for security arose when doing security consulting for a project during a research visit with M. Abadi at Bell Labs (Lucent Tech.), Palo Alto, whose hospitality is gratefully acknowledged. This work benefitted from discussions at the summer school "Foundations of Security Analysis and Design 2000" (in particular, Li Gong suggested to apply the UML-based approach to security to guarded objects) and the Dagstuhl seminar "Security through Analysis and Verification" (in particular with D. Gollmann and B. Pfitzmann).

The work was presented in two talks at the Computing Laboratory at the University of Oxford. Comments from S. Abramsky, C. Crichton and G. Lowe are gratefully acknowledged. Finally, comments by the anonymous referees have been very helpful.

References

[Aba00] M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*. IOS Press, 2000.

[ABLP93] M. Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[BV99] B. Bokowski and J. Vitek. Confined types. In *14th Annual ACM SIGPLAN Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA'99)*, 1999.

[CCR01] R. Campo, A. Cavarra, and E. Riccobene. Simulating UML state machines. In E. Börger and U. Glässer, editors, *ASM'2001*, LNCS. Springer-Verlag, 2001. To be published.

[FDR94] J. C. Fabre, Y. Deswarte, and B. Randell. Designing secure and reliable applications using fragmentation-redundancy-scattering: an object-oriented approach. In *PDCS 2: Open Conference*, pages 343–362, Newcastle-upon-Tyne, 1994. Dept of Computing Science, University of Newcastle, NE1 7RU, UK.

[GAS99] Stefanos Gritzalis, George Aggelis, and Diomidis Spinellis. Architectures for secure portable executable content. *Internet Research*, 9(1):16–24, 1999.

[Gol99] D. Gollmann. *Computer Security*. J. Wiley, 1999.

[Gon98] Li Gong. Java^{TM} Security Architecture (JDK1.2). http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html, October 2 1998.

[Gon99] Li Gong. *Inside Java 2 Platform Security – Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[GSG99] Stefanos Gritzalis, Diomidis Spinellis, and Panagiotis Georgiadis. Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification. *Computer Communications Journal*, 22(8):695–707, 1999.

[HKK00] Manfred Hauswirth, Clemens Kerer, and Roman Kurmanowytsch. A secure execution framework for Java. In *ACM conference on Computer and communications security*, 2000.

[Jür01a] Jan Jürjens. Developing secure systems with UMLsec — from business processes to implementation. In *VIS 2001*. Vieweg-Verlag, 2001. To appear.

[Jür01b] Jan Jürjens. Encapsulating rules of prudent security engineering. In *International Workshop on Security Protocols*, LNCS. Springer-Verlag, 2001. To be published.

[Jür01c] Jan Jürjens. Modelling audit security for smart-card payment schemes with UMLsec. In M. Dupuy and P. Paradinas, editors, *Trusted Information: The New Decade Challenge*, pages 93–108. International Federation for Information Processing (IFIP), Kluwer Academic Publishers, 2001. Proceedings of SEC 2001 – 16th International Conference on Information Security.

[Jür01d] Jan Jürjens. *Principles for Secure Systems Design.* PhD thesis, Oxford University Computing Laboratory, 2001. In preparation.

[Jür01e] Jan Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (International Symposium)*, volume 2021 of *LNCS*, pages 135–152. Springer-Verlag, 2001.

[Jür01f] Jan Jürjens. Towards development of secure systems using UMLsec. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering (FASE/ETAPS, International Conference)*, volume 2029 of *LNCS*, pages 187–200. Springer-Verlag, 2001. Also OUCL TR-9-00 (Nov. 2000), http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-9-00.html.

[Kar00a] G. Karjoth. Authorization in CORBA security. *Journal of Computer Security*, 8(2,3):89–108, 2000.

[Kar00b] G. Karjoth. Java and mobile code security – an operational semantics of Java 2 access control. In *IEEE Computer Security Foundations Workshop*, 2000.

[KG98] L. Kassab and S. Greenwald. Towards formalizing the Java Security Architecture in JDK 1.2. In *European Symposium on Research in Computer Security (ESORICS)*, LNCS. Springer-Verlag, 1998.

[LGK$^+$99] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the Java platform. In *IEEE Annual Computer Security Applications Conference*, 1999.

[Mea01] W. Measor. Secure byzantine agreement – design, implementation and verification. Master's thesis, Oxford University Computing Laboratory, 2001.

[ND97] V. Nicomette and Y. Deswarte. An authorization scheme for distributed object systems. In *IEEE Symposium on Security and Privacy*, 1997.

[RSG$^+$01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2001.

[Sam00] P. Samarati. Access control: Policies, models, architectures, and mechanisms. Lecture Notes, 2000.

[SP00] P. Stevens and R. Pooley. *Using UML.* Addison-Wesley, 2000.

[SS75] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SS94] R. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9), 1994.

[UML01] UML Revision Task Force. OMG UML Specification v. 1.4 (draft). OMG Document ad/01-02-14. Available at http : //www.omg.org/uml, February 2001.

[Var95] V. Varadharajan. Distributed object system security. In H.P. Eloff and S.H. von Solms, editors, *Information Security - the next Decade*, pages 305–321. Chapman & Hall, 1995.

[VH96] V. Varadharajan and T. Hardjono. Security model for distributed object framework and its applicability to CORBA. In *12th International Information Security Conference IFIP SEC'96*, 1996.

[WF98] D. Wallach and E. Felten. Understanding Java Stack Inspection. In *IEEE Security and Privacy*, 1998.