Trust-adapted enforcement of security policies in distributed component-structured applications*

Peter Herrmann and Heiko Krumm
Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund
{herrmann|krumm}@ls4.cs.uni-dortmund.de

Abstract

Software component technology on the one hand supports the cost-effective development of specialized applications. On the other hand, however, it introduces special security problems. Some major problems can be solved by the automated run-time enforcement of security policies. Each component is controlled by a wrapper which monitors the component's behavior and checks its compliance with the security behavior constraints of the component's employment contract. Since control functions and wrappers can cause substantial overhead, we introduce trust-adapted control functions where the intensity of monitoring and behavior checks depends on the level of trust, the component, its hosting environment, and its vendor have currently in the eyes of the application administration. We report on wrappers and a trust information service, shortly outline the embedding security model and architecture, and describe a Java Bean based experimental implementation.

Key Words: software components, wrappers, trust management, security policy enforcement, trust information service.

1. Introduction

As enterprises are increasingly dependent on their information systems, the security of the information systems is of growing importance. Therefore security models are designed identifying the relevant principals, objects, operations, attributes, and relations of the systems on an abstract user-oriented level. Security policies refer to the model and express the different security objectives to be enforced. The objectives concern confidentiality, integrity, accountability, and availability properties of data and functions as they are demanded by the different principals. With respect to the systems' implementation and operation security architectures are developed. A security architecture reflects the special architecture of an information system and defines the types, positions, and applications of security services to be integrated into the system in order to provide the security objectives even under presence of attacks. The services are implemented by suitable combinations of security mechanisms. Current application middleware platforms already adopted this view. They facilitate the corresponding provision of security and supply suitable models and services. So, e.g., the widely used CORBA platform meanwhile supports secure distributed object-based application systems.

The approach of component-structured software envisages applications composed from cost-effective components. The components are supplied by different developers and are offered to a growing community of customers on an open market (cf. [30]). By selection, configuration, and customization of components powerful applications can be built which are tailored to the special needs of single customers. Their architecture reflects very flexibly the user requirements and their environment. The applications are easily extensible and modifiable by dynamic changes of components and their coupling. Moreover, since applications are built by defining the communication and coordination of components, the same means can be used to integrate different applications to cooperating super-applications.

Meanwhile a series of platforms supports components. Most prominent are Java Beans and Enterprise Java Beans. The COM/DCOM approach is well-established in PC-based environments. Moreover, the CORBA initiative extended its approach to the comprehensive support of component structures. The platforms typically provide notions for the description of component types, parameter types, and interfaces. They supply rich run-time support for the coupling of components and, in particular, enable introspection, the exploration of components, their interfaces, and their properties at run-time. Additional interest is given to the comfortable construction of applications by scripting languages or visual application builder tools.

The expected benefits of software component technology as well as the increasing availability of powerful mid-

*This work was funded by the German research foundation DFG.

1530-1346/01 $10.00 © 2001 IEEE

dleware platforms plead for a rapidly growing employment of component-structured distributed applications. The architecture of those application systems, however, really extends the architecture of distributed object-based applications. In particular it imposes new security aspects since it introduces new principals and roles. In addition to users and application system owners, also component vendors and host providers have to be considered. On the one hand they introduce their own security objectives. On the other hand, they introduce new types of threats since in general the different principals cannot trust each other to full extent.

Of course, the composition of applications from various components causes not only security problems. Among other properties, in particular it is essential for the functionality of an application that each component acts in accordance with its specifications. Therefore the approach of software components refers to the employment of explicit contracts. Each component integration is accompanied by a contract describing the agreed properties of the component, especially its interface, operations, and the relations with its environment (cf. [30]).

Our overall approach is also based on contracts. A component contract has to contain a description of the security-relevant behavior with which the component's execution is assumed to correspond. At design time, the structure of the system is analyzed in combination with the behavior descriptions of its components in order to prove that required security properties of the system hold if each component will act in accordance with its contract. At run-time, the consideration can focus on the components. For each component it is of interest that its actual behavior in fact is conformable with its contract since malicious components or compromised code (e.g., by virus or Trojan horse infection) will result in diverging behaviors. For that purpose the component behavior can be controlled at run-time by means of wrappers. Detailed control functions, however, can cause substantial overhead. Therefore we propose the dynamic adaptation of the control functions to that level of trust, the component currently can have in the eyes of the application owner.

In the sequel we will only outline the overall approach and concentrate on the trust-adapted security-policy enforcement which is implemented by component wrappers and a trust-managing infrastructure. Firstly we sketch the major security aspects of distributed component-structured software. Thereafter we refer to related work concerning the enforcement of security policies and the management of trust relations. The next section describes our special component control approach and explains wrappers, trust-management, and their interactions. Later on we discuss the trust management system in more detail and report on a Java Bean based implementation of the approach.

2. Security aspects of components

One can identify a hierarchy of application architecture classes which starts with local applications, comprises distributed and mobile code applications, and ends with distributed component-structured applications. The security aspects of local applications focus on the definition of user classes, on their authentication, and on access control since trusted software and a trusted hosting system are assumed. The security of distributed software additionally has to recognize that parts of an application reside on different host computers which are connected by a communication network. Though the hosts mostly are trusted, the network is not. The binding between application parts as well as their communication may be attacked via the network. Therefore services for the secure communication are needed. Moreover, authentication and access control must be able to operate in a distributed environment. In a further extension mobile code applications contain parts which can migrate between hosts. Here also the secure transfer of code modules has to be provided. Moreover, since host and application owners not necessarily trust each other, host environments need protection against malicious code and vice versa the applications may be attacked by malfunctions of hosts (cf. [10, 18]).

Distributed component-structured applications can consist of software components which are supplied by different vendors. Therefore one has to distinguish between application owners and software component vendors and there is a needs for corresponding protection:

- Protection of an application against malicious components with respect to the integrity of the application, its configuration, and its resources, to the confidentiality and availability of managed information and supported operations, as well as to the accountability of performed actions.
- Protection of a component vendor against wrong incriminations with respect to spiteful application administrators, malicious surrounding components, and malfunctions of hosting environments. Moreover a component vendor is interested in protection against unlicensed employment of components.

The corresponding security model identifies a rich set of principals comprising users, resource owners, application owners, host providers, and component vendors. The main objects of the model are resources, software components, application configurations, hosting environments, and communication facilities. The major relations concern that components constitute applications and contribute to the functionality of an application. So, a component accesses and manages resources on behalf of a user. Moreover, it forwards information and control to other components and to

the application's environment using several types of component interface mechanisms (e.g., component attributes, events, use of special supportive services like life-cycle, binding, and configuration services, "normal" method invocations and returns). During run-time a series of interface mechanism events occurs at the border of a component which forms its behavior. A component-specific security policy can constrain this behavior by defining static as well as dynamic conditions.

The corresponding security architecture is based on the extension of each component employment contract by a set of static and dynamic behavior constraints expressing the security policy of the component's employment. Control functions perform a monitoring of the component's interfaces. They dispose of a run-time representation of the behavior constraints and enforce the constraints. Additionally applications contain manager components which interact with the control functions. A manager component supervises the control functions, it activates and deactivates them, and receives their notifications. Moreover, it supplies an administration interface and communicates with external services. There is a needs for two types of external services:

- Trust information service,
- Conflict resolution service.

A trust information service records component vendors, component employers, component types, and reports about the employment of components. It calculates the accumulated risks of component types. It considers the positive and negative reports on components and evaluates the current component trust levels. It communicates trust information on demand and notifies registered applications about significant changes.

Mainly, a conflict resolution service has to resolve conflicts between component vendors and application owners about actual or supposed breaches of employment contracts. Moreover, conflicts between trust information services and component vendors have to be handled if vendors complain of wrong bad evaluations of their products. Also, application owners may complain of getting wrong good or delayed alert information from a trust information service. Vice versa a trust information service may complain of getting wrong information from application owners or component vendors. Of course not all conflicts can be resolved by automated services without human interaction. The services, however, can prepare external decisions. So, they may manage the integration of additional log collecting components into application configurations.

3. Related work

Security of component-structured software benefits from the research done in the field of mobile code. In this field various approaches were recently developed in order to protect host computers against attacks by mobile programs. These methods mainly focus on control flow safety, memory safety, and stack safety [20]. Besides of isolating security-critical operations in a protected system kernel (e.g. [2]) and using cryptography for the transit of code, code instrumentation gained attraction in the last years. Here, machine code is altered in a way that critical operations can be analyzed before or monitored during the execution of the code in order to detect attacks. An example is software fault isolation (eg., [32]) where non-trusted code is executed and monitored in a safe system part where it cannot cause damage.

Another approach based on code instrumentation is followed by Schneider [28, 29] who models policies formally by so-called security automata. Moreover, a security automaton can be used to enforce a policy by simulating it simultaneously to the execution of the code. The code is only permitted to perform an execution step if that corresponds to a transition of the automaton. The automata based enforcement extends the early approach of state dependent security specifications [4].

Language based security is a kind of code instrumentation, too. Here, special security-related information about mobile code is obtained during parsing or other program analyses. The program user utilizes this information in order to check the code for compliance with his security policies. An example is the Java byte code verifier which proves Java byte code for type correctness and other security-related properties. Another method is proof carrying code (cf. [20]) which enables formal program verification. The program developer annotates the code with a formal specification (fi., pre- and postconditions of functions or loop invariants) and hands this information over to the user who proves the code formally. Examples for utilizing proof carrying code are the touchstone compiler [27] and the efficient code certification [19]. Moreover, this method was used for more specialized verification purposes as type checking [24, 31] and information flow analysis [11, 25, 26].

Since the information used for code verification is produced by the code developer, it may be distorted in order to mask malicious code. Thus, one has to check that the program complies to the additional information used for verification. Here, the concept of generic software wrappers proves helpful. In this approach a program is extended by a software checking the code during runtime for security properties without changing it. Generic software wrappers are used with firewalls and intrusion detection [1, 13, 23]. Moreover they can also be applied for protecting component-structured software performing malicious system calls [12].

Another field of interest for our approach is trust management. Here, the trust in a human or computer principal

is expressed mathematically depending on good and bad experiences with this principal. In [16], Jøsang introduces a so-called opinion-triangle expressing belief, disbelief, and uncertainty about a principal. These values can be calculated from the number of good and bad experiences with the principal by metrics. Jøsang and Knapskog [17] calculate the belief in a principal as the ratio of the number of positive experiences and the total number of positive and negative experiences. Thus, a negative experience can be compensated by some positive experiences. In contrast, in Beth's et al. [3] metric a single negative experience leads to complete disbelief in a principal forever. If no bad experience occurred, the belief grows with an increasing number of good experiences.

Trust management is used for authorization systems [6]. Unlike classical mechanisms like access control lists, the access to an action or a resource is provided if the caller shows a certain number of credentials issued by third parties which are trusted by the server. Implementations of trust management-based authorization systems are PolicyMaker [7], REFEREE [8], and KeyNote [5].

4. Trust-adapted enforcement

One dominating security problem of software component employment is that the component code may not be trusted to full extent. Therefore techniques are of interest by means of which the application owner can analyze the compliance between component code and component specification. Our proposal relies on simulated security automata [28] since they support state dependent security constraints [4] and can manage the analysis without difficult explicit formal verification. The security automata are automatically integrated by means of generic wrappers [23]. In combination with additional measures run-time behavior checks are of further use and contribute to the control of non-trusted hosting environments.

A simple example shall exemplify the conception of state dependent security constraints. There may be two components in an application system, a component *AC* controls the access to a set of resources, and a component *OP* performs operations on the resources. It is intended that *OP* only accesses resources by means of a resource handle which *OP* has formerly obtained from *AC* on behalf of a user. This can be expressed by a state dependent constraint for the behavior of *OP*. The corresponding state automaton has a state space of type "*set of pairs of resource handle and user identification*". The initial state is the empty set. Whenever an event of type "*AC grants a resource handle rh for user us*" occurs, a transition is performed which inserts the pair $\langle rh, us\rangle$ into the current state. Whenever an event of type "*OP closes a resource handle rh*" occurs, another transition deletes the corresponding pair from the state. Moreover there may be a transition which spontaneously removes pairs after a specified lifetime is exceeded. Finally all events of type "*OP accesses a resource under* $\langle rh, us\rangle$" are connected with a transition which is combined with the security condition "$\langle rh, us\rangle$ *is element of the current state*". Though this example is simple, it shows how state dependent security constraints can help to manage the distribution of security responsibilities between run-time components.

As shown in the example state dependent security constraints can be modeled by state automata and can be checked at run-time by automata implementations which are linked with the controlled behavior. Each significant event of the controlled behavior is linked with a corresponding transition type. Thus an automaton records the history of events and pending events can be checked for compliance: If an event is pending, the corresponding transition must be enabled.

This principle, however, has to be extended in order to support dynamic adaptation of behavior control. Since it is possible to provide a series of logically AND-connected behavior constraints for the same component, adaptation can be performed by dynamic activation and deactivation of single constraints. In the implementation the constraints correspond with a series of state automata where all automata are simulated simultaneously. Therefore, in a first setting, adaptation can be performed by activating and deactivating the automata implementations. Nevertheless, this simple solution works only in the special cases where the state of an automaton can be initialized on activation or where the events occurring during deactivated periods are not relevant for later activated periods. In the example above this is not the case. *OP* may obtain a valid handle during a deactivated phase and use the handle in a following activated phase.

In the general case, an automaton cannot be deactivated in the whole. We therefore propose to structure the activities of a transition into three parts:

1. Check of the enabling condition,
2. Check of the security condition,
3. Computation and assignment of the next state.

The parts 1 and 3 are not subjected to deactivation. They are designed to record the relevant event history. So, in the example above, insert and remove transitions can keep track of the set of valid resource handles. Only the execution of part 2 — which is devoted to the computation of (sometimes more complex) state checks — depends on the current activation state of the automaton. In the example resource access transitions are accompanied by a security condition checking the validity of handles.

The conception of automata is also used for the specification of state dependent security constraints. Therefore component employment contracts contain automata definitions

which as well correspond to the logically interesting security specifications as they enable the direct translation to run-time checking automata simulations. Mostly, the security specifications are developed by a component producer and handed over to the component user. Before the corresponding employment of a component the user checks the specifications for compliance with the security policies of the system to which the component will belong.

In our more special approach we specify the security constraints in the formal specification technique cTLA [14, 15] which is based on Leslie Lamport's Temporal Logic of Actions (TLA) [21]. cTLA facilitates the description of safety, liveness, and real-time properties in a process-like style similarly to programming languages. In particular it supports the definition of so-called constraint-oriented processes which can model single aspects of components. Thus, different security properties (e.g., different information flow and data access restrictions) may be specified by separate cTLA processes.

Moreover one can apply cTLA for the specification of abstract system security policies and one can combine component specifications to formal system specifications. This forms the basis for formal cTLA based system proofs. They check if an application system under given composition and component contracts will comply with the abstract policies. The proofs can be performed by formal logical reasoning and will be explained elsewhere.

5. Trust information service

In order to reduce the performance expenditure of a security wrapper, positive and negative experiences, other users gained with the wrapped component, can be utilized. A component user may trust a component based on these experiences to a certain extent and may vary the intensity of safeguards based on the level of trust. For instance, a component should be thoroughly scrutinized for launching attacks, if it is not known very well or already attacked other users. If it, however, is well known and never caused bad experiences, the safeguards can be reduced to spot checks or omitted at all.

In our approach the management of trust is performed by a trust information system (cf. Fig. 1). It consists of a trust manager for each security wrapper, a trust information service, and a cipher service. The trust managers determine the intensity of security checks by their wrappers depending on the trust values carried by the scrutinized components.

The trust information service stores reports about positive and negative events and calculates trust values from them. For determining a trust value we currently use a combination of the metrics of Jøsang [16] and Beth et al. [3]. The events come from the trust managers which indicate in intervals their experiences with the checked component.

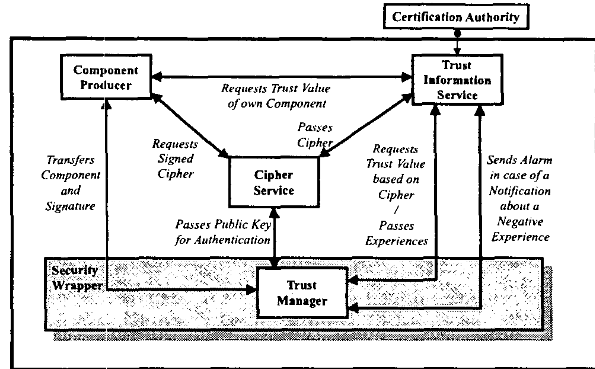

Figure 1. Trust Information System

Moreover, the reports of product certification authorities can be used for the calculation of trust values as well. The trust manager requests the trust value of an interesting component in order to decide the level of security checks. Furthermore, the trust information service offers an alarm service notifying all interested trust managers immediately about a negative event report of a particular component. Additionally, trust information services may provide typical consumer information which can prepare purchase decisions.

In order to guarantee the privacy of component producers, the trust information service can use ciphers instead of complete component descriptors. The ciphers are created by the cipher service. If a component producer wants to register a component, it does not call the trust information service but the cipher service which generates a cipher and a signature consisting of the cipher and a component signature passed by the producer. Thereafter, this signature is handed over to the producer while the trust information service receives only the cipher. When the producer sells the component, it passes the signature to the customer. A trust manager of the customer authenticates the cipher of the component by means of the signature. Afterwards, it interacts with the trust information service using only the cipher instead of the full component name. The privacy is guaranteed since the trust information service knows only the ciphers and not the real component names, the trust managers use only trust values of a very limited number of components, and the cipher service has no knowledge about trust values.

6. Wrapper for Java Beans

A first implementation of our approach is centered around the component model of Java Beans [22]. The enforcement system is sketched in Fig. 2 and consists of an adapter for each Java Bean to be checked, a number of ob-

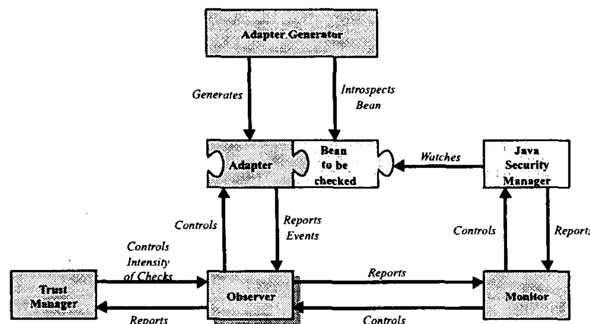

Fig. 2: Architecture of the Security Wrapper

Figure 2. Architecture of the Security Wrapper

servers, a trust manager, a monitor, an adapter generator, and the Java security manager.

In order to scrutinize the behavior at the component interface, an adapter is wrapped around the bean. The bean does not interact with its environment directly but only via the adapter. Thus, the events at the interface can be observed. Moreover, the adapter can seal the bean if it is suspected to attack its environment. It wraps incoming and outgoing calls of Java Bean events, properties, methods, and exceptions as well as messages of an Infobus.

The check that a bean complies with cTLA process specifications is performed by the observers. Each observer corresponds to one cTLA constraint process. It contains the state representation of the process. Furthermore it provides an initialization method and action methods. The action methods test the enabling condition and the security condition of the corresponding action. Moreover they perform the state transitions. The initialization method resets the state representation.

If the adapter detects an interface event, it forwards it to the observers by calling the corresponding action methods. If all relevant observers signaled the compliance of the event, the adapter really transfers the interface event. If, however, one observer refuses its corresponding action, the adapter seals the bean and reports the violation to the monitor. The monitor will inform the system administrator who has to decide about further actions. Moreover, the observers report violations also to the trust manager which informs the trust management service about a negative event and controls the intensity of checks. It activates and passivates the observers depending on the current trust levels of the components.

The system administrator performs the overall supervision of the system including the security wrappers by means of the administration monitor. The monitor supports an interactive interface to the system administrator. It forwards reports from the observers and from the Java security manager to the administrator and accepts control commands of the administrator. Prior to the employment of a bean the administrator has to create an adapter by means of the automated adapter generator. This generator introspects the bean for its interface actions and creates the related adapter based on this examination.

Additionally to policy enforcement, the approach utilizes the built-in Java security manager. A bean does not only communicate with its environment via the component interface but can also access system resources directly using streams. These stream-based accesses, however, cannot be detected by the introspection of the adapter generator and therefore are not controlled by the adapter. Nevertheless, system resource accesses have to comply to the Java security manager. Therefore the security manager was extended by functions to report accesses to the monitor. Moreover, the administrator can decide which resource accesses are allowed or forbidden.

The approach was tested with a component-structured warehouse management system for franchise fast food restaurants [22]. This application was created with the SalesPoint framework [9] containing a set of Java objects to create business applications. We transferred a part of the framework objects to Java beans. The test was performed with three beans which realize a catalogue of goods offered in the restaurant, the counting stock of these goods, and the management functions. We considered the catalogue bean as not secure since it offers a direct link to the franchise company in order to fix prizes, input new offers, etc. The wrapper checked that this bean does not access the counting stock in order to prevent the franchise company to violate the privacy of the restaurant owner by wiretapping information about product sales. For testing purposes we used various versions some of which attacked the counting stock. The wrapper worked correctly and detected all violations. The mean value of the runtime overhead in comparison to running the bean without a security wrapper was 4.79% on a Pentium 200 PC with Windows NT. Most of the overhead was caused by the graphical interface of the administration monitor. Without the monitor, the overhead was just 0.34% in the example.

7. Concluding remarks

We proposed the approach of trust-adapted enforcement of security policies which can be of high interest for future complex multi-vendor component-structured applications. The embedding security architecture has to consider various other security aspects as well and is not completely elaborated yet. Currently we deal with the design of a more detailed security model which, e.g., shall consider host employment contracts and shall integrate existing approaches

for secure distribution and code motion. A series of discussions and future contributions of others may be necessary to complete this model.

References

[1] F. M. Avolio and M. J. Ranum. A Network Perimeter with Secure External Access. In *Proc. Internet Society Symposium on Network and Distributed System Security*, Glenwood, 1994.

[2] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. 15th Symposium on Operating System Principles*, pages 267–284. ACM, 1995.

[3] T. Beth, M. Borcherding, and B. Klein. Valuation of Trust in Open Networks. In *Proc. European Symposium on Research in Security (ESORICS)*, LNCS 875, pages 3–18, Brighton, 1994. Springer-Verlag.

[4] J. Biskup and C. Eckert. About the enforcement of state dependent security specifications. In T. Keefe and C. Landwehr, editors, *Database Security*, pages 3–17. Elsevier Science (NorthHolland), 1994.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System, Version 2. Report RFC-2704, IETF, 1999.

[6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The Role of Trust Management in Distributed Systems Security. In J. Vitek and C. Jensen, editors, *Internet Programming: Security Issues for Mobile and Distributed Objects*. Springer-Verlag, 1999.

[7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. 17th Symposium on Security and Privacy*, pages 164–173, Oakland, 1996. IEEE.

[8] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. *World Wide Web Journal*, 2:127–139, 1997.

[9] B. Demuth, H. Hussmann, L. Schmitz, and S. Zschaler. A Framework-based Approach to Teaching OOT: Aims, Implementation, and Experience. University of the Armed Forces, Munich, 2000.

[10] W. Farmer, J. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proc. 19th National Information Systems Security Conference (NISSC 96)*, pages 591–597, 1996.

[11] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, 1997.

[12] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proc. 1999 IEEE Symposium on Security and Privacy*, 1999.

[13] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proc. 6th USENIX Security Symposium*, 1996.

[14] P. Herrmann and H. Krumm. Formal Hazard Analysis of Hybrid Systems in cTLA. In *Proc. 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99)*, pages 68–77, Lausanne, 1999. IEEE Computer Society Press.

[15] P. Herrmann and H. Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[16] A. Jøsang. An Algebra for Assessing Trust in Certification Chains. In J. Kochmar, editor, *Proc. Network and Distributed Systems Security Symposium (NDSS'99)*. The Internet Society, 1999.

[17] A. Jøsang and S. J. Knapskog. A metric for trusted systems. In *Proc. 21st National Security Conference*. NSA, 1998.

[18] G. Karjoth, D. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, pages 68–77, July/August 1997.

[19] D. Kozen. Efficient code certification. Technical Report 98–1661, Computer Science Department, Cornell University, 1998.

[20] D. Kozen. Language-Based Security. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proc. Conference on Mathematical Foundations of Computer Science (MFCS'99)*, LNCS 1672, pages 284–298. Springer-Verlag, 1999.

[21] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[22] A. Mallek. Sicherheit komponentenstrukturierter verteilter Systeme: Vertrauensabhängige Komponentenüberwachung. Diplomarbeit, Universität Dortmund, Informatik IV, 44221 Dortmund, 2000 (in German).

[23] M. A. Monroe. Security Tool Review: TCP Wrappers. *;login:*, 18(6):15–16, 1993.

[24] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, 1998.

[25] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, 1999.

[26] A. C. Myers and B. Liskov. Complete, Safe Information with Decentralized Labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, 1998.

[27] G. C. Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, 1998.

[28] F. B. Schneider. Towards fault-tolerant and secure agentry. In *Proc. 11th International Workshop on Distributed Algorithms (WDAG'97)*, LNCS 1320, pages 1–14. ACM SIGPLAN, Springer-Verlag, 1997.

[29] F. B. Schneider. Enforcable Security Policies. Technical Report TR98–1664, Computer Science Department, Cornell University, 1998.

[30] C. Szyperski. *Component Software — Beyond Object Oriented Programming*. Addison-Wesley Longman, 1997.

[31] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. Conference on Programming Language Design and Implementation*. ACM SIGPLAN, 1996.

[32] R. Wabbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th Symposium on Operating System Principles*, pages 203–216. ACM, 1993.