A Survey of Some Implementation Techniques for Security Membranes [⋆]

Marc Lacoste

Distributed Systems Architecture Department, France Télécom R& D.
marc.lacoste@rd.francetelecom.com

Abstract. The notion of security membrane appears as an emerging concept in the design of secure languages for global computing. Membranes separate the computational behavior of a site from the security code controlling access to site-located resources. We provide a survey of some of the challenges which arise when trying to implement security membranes in an execution platform such as an operating system. We identify four main design issues: the choice of a security model; the type of architecture for the execution environment; the layer at which to place security mechanisms; and the assurance level of the platform. In each case, we discuss possible trade-offs between security, flexibility, simplicity, and trustworthiness. We then show how applying a component-based approach to design and implement the execution environment can help to reach an acceptable compromise between such properties.

1 Introduction

The notion of *security membrane* is increasingly attracting attention for the design of secure languages for global computing environments, as witnessed by a number of recent papers [11, 23, 38, 45]. The aim of a membrane is to clearly separate the computational entities of a location, generally modeled as a number of processes running in parallel, from a generic controller, the *membrane* (also called *guardian*) which supervises the behavior those entities at run-time, and acts as a unique gateway with the outside of the location [11, 45]. Thus, a location named ℓ can be modeled as $\ell\ (\ M\ )\{\ P\ \}$, where M is a specification of the membrane behavior, and P is a parallel composition of processes. A membrane should be seen as an enhanced container that can provide fine-grained control over the semantics of communication or mobility for localized processes.

A membrane can also be used for security-specific purposes to protect the location, by filtering incoming and outgoing messages. This allows to realize a security domain, by selectively controlling access to a group of processes which share a common security policy. This policy is often expressed as a type system [13, 16, 24, 28], against which processes will be type-checked, either statically or dynamically, to ensure safe behavior. Location ℓ can then be written $\ell\ (\ \pi\ ,\ RM\ )\{\ P\ \}$, where π is a set of types defining the location security policy, and *RM* is a *reference monitor* [2], abstract machine in charge

[⋆] This work has been partially supported by EU FET Global Computing initiative, project MIKADO IST-2001-32222.

of enforcing the authorization policy π. In general, a clear separation is made between the programmer who implements the process behaviors, and the network administrator who defines the security policy, for a site or a group of sites.

So far, research on the protection of location resources in a global computing environment has mainly focused on language-based security, for instance by defining numerous type systems for distributed process calculi with locations. But is this enough? What is needed in practice to implement a security membrane? In existing approaches, little is said about effective implementation requirements. The emphasis is put on proving theoretical properties, like type safety results, leaving enforcement in an execution platform to traditional code verification techniques, like model checking or proof-carrying code [41]. Yet, some of these methods still lack maturity. For instance, most automated verification tools for mobile code, like model-checkers and theorem provers, reach their limits for large-size systems, and a human intervention is often needed to carry out the proof of safety. Still, providing strong end-to-end security guarantees in software infrastructures is critical to protect assets located within network locations against unauthorized disclosure and improper modification, while ensuring service availability to legitimate users. Such security requirements call for strong protection mechanisms down to the lowest software layers like the operating system, which serves as a foundation to guarantee security at higher levels.

We address this problem by studying in this paper what are the practical implementation challenges for the realization of a security membrane, encompassing not only the language aspects, but also the platform-related issues, in particular in the operating system. The objective is to acquire an overall perspective of some of the critical design parameters for the effective implementation of such a membrane.

A first sensible design principle for controlling access to location resources is to clearly separate the security policy from the enforcement mechanisms. The former captures the protection requirements to be satisfied, for instance gathered after a vulnerability analysis. The latter should be understood as a Trusted Computing Base (TCB), defined in the Orange Book [50] as "the totality of protection mechanisms within a computer system – including hardware, firmware and software – the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system." Hence the need for a bird's eye view of security membrane design, which should not be limited just to language issues.

The above separation enables access control mechanisms to be reused to enforce different security policies. As such, it has a direct impact on the overall flexibility of the security membrane. This requirement is also motivated by the need to federate several authorization policies in global computing networks, seen as superpositions of multiple heterogeneous subsystems, each endowed with its own security policy.

A membrane should (at least) guarantee the following properties:

Security: no illegitimate access to resources should be possible; access control mechanisms should guarantee complete mediation, i.e., they should be non-circumventable; they should also be tamper-proof, to preserve integrity; finally, the principle of least privilege should be enforced to avoid abusive propagation of delegated access rights.

Flexibility: the membrane should support multiple authorization policies, and allow fine-grained access control; it should also enable dynamic management of access rights to deal with unpredictable changes in the environment of the location; finally, it should provide mechanisms to manage the delegation of privileges.

Performance: the impact of security checks on the performance of running processes should be unobtrusive, or, at least, acceptable.

Simplicity: a principle of economy of mechanism should guide the membrane design, so that it may be simple to implement, to use, and to administer.

Trustworthiness: the membrane implementation should be small enough to be subject to analysis, in order to provide evidence to independent evaluators like trusted-third parties that the protection mechanisms fulfill in a correct way the identified security requirements for the location.

It seems difficult to find the right balance between such often conflicting properties. We identify the following important criteria in the design of security membrane, which may help to reach a compromise:

– *The choice of the security model:* what security property (i.e., confidentiality, integrity, least privilege, etc.) is the membrane trying to enforce?
– *The type of operating system kernel:* operating systems architecture has greatly evolved from monolithic and micro-kernels to more extensible systems. Which architecture allows to reach the best compromise between the properties mentioned above?
– *The choice and location of security mechanisms:* which protection mechanisms should be used? Is language-based security sufficient? Should the membrane also rely on operating system protection, or even on hardware security? What are the right levels of abstraction and the appropriate software layers to place security mechanisms in the system?
– *The level of assurance of the platform:* what guarantees can the platform provide of correct policy enforcement? What metric should be used to assess the strength of the security membrane? How trustworthy really is the access controller?

The paper provides an overview of the involved trade-offs for each key design issue identified previously. Based on that quick survey, we then consider a possible solution for the architecture of security membranes, by applying a component-based approach to its design. We argue that this approach enables to build secure yet adaptable security membranes, where flexible access control mechanisms can be easily designed and implemented, allowing support for multiple authorization policies.

The rest of the paper is structured as follows: we first give a survey of possible trade-offs between security, flexibility, performance, simplicity, and trustworthiness for a security membrane, regarding: the choice of a security model (section 2), the type of architecture for the execution environment (section 3), the layers at which to place security mechanisms (section 4), and the platform assurance level (section 5). We then discuss in section 6 the benefits of a component-oriented design for implementing flexible security membranes.

2 Choosing a Security Model

A security model is an abstract description of a security policy, which forms the theoretical setting for the expression of the policy, and provides a basis for reasoning to prove security properties.

2.1 A Brief Overview of Existing Security Models

Overcoming the Safety Problem. The *access control matrix* [34] is a foundational model describing the protection state of an authorization system. Unfortunately, proving system safety with this model was shown to be undecidable [27]. To overcome this difficulty, the focus of recent research on access control models has shifted towards the following directions:

– The study of less expressive models, which can be implemented more easily, and on which reasoning becomes possible. This restriction is at the expense of flexibility, since in those limited models, security policies are generally static, i.e., the assignment of access rights from subjects to objects cannot vary in time, or has to be done by trusted principals [3, 4].
– The extension of the access matrix with *constraints* to facilitate the verification of security properties. Those models also capture more dynamic aspects of a security policy, like the changing of privileges over time [12, 22, 47]. Unfortunately, constraints-based models are hard to implement due to the complexity of the logical languages used to express the constraints.

Thus, a compromise has to be reached between: the expressiveness of the security model, which is a first metric of its genericity and flexibility; the simplicity of implementation; and the easiness of enforcing and verifying security properties within the model.

Which Security Policy? Access control policies may be classified according to the security properties one wishes to prove on a system. Discretionary access control schemes, like those found in traditional UNIX security, reach their limits for systems with strong security requirements. A mandatory approach to security is then needed, so that authorization mechanisms may not be circumvented. The covered spectrum is quite broad, since the explored concepts range from simple properties like confidentiality [4] and integrity [9, 10], to more advanced principles like least privilege and separation of duties [12, 22, 47]. For instance, see [7] for a detailed comparison of some access control models and their expressive power.

Confidentiality Policies. A number of lattice-based models, like the one from Bell and Lapadula [4], focus on preventing unauthorized disclosure of information, notably for multi-level military-grade security systems. These models enable extended analysis of information flows.

Integrity Policies. Other models try to preserve the integrity of systems. For instance, the Biba integrity model [9] is the exact dual of the Bell-Lapadula one, where security levels classify the integrity of information instead of its confidentiality. Clark and Wilson [14] provide a model more adapted to commercial environments by introducing integrity constraints. Boebert and Yain [10] study non-hierarchical mandatory integrity policies, and introduce the notion of *type-enforcement*, used for instance in the DTE (Domain and Type Enforcement) model for UNIX security [3]: *types* are assigned to resources, and processes are grouped into *domains*, which grant a number of access rights over specific types.

Least Privilege Policies. This family of policies tries to enforce the concept of *separation of duties* (SoD): each task in a sequence of operations has to be performed by a different subject [47]. A variant is the Chinese Wall [12] policy, where a history of accesses to resources is kept to resolve conflicts of interests in commercial systems. A formal study of different types of least privilege policies can be found in [22].

Role-Based Policies. By introducing the abstraction of *role*, the RBAC (Role-Based Access Control) model [44] makes administration of access rights easier. Users are assigned *roles* which confer permissions over a set of resources. This model can represent several classes of security policies [42], and is a first step towards policy-neutral access control (cf. section 2.2). The price to pay is a greater complexity: for instance, the NIST RBAC standard [19] distinguishes no less than four variants of the model, such as *flat* RBAC, which defines assignments of roles to users and of permissions to roles, *hierarchical* RBAC where roles are structured into a hierarchy, and *constrained* RBAC which enables expression of additional constraints to enforce separation of duties.

2.2 Towards Policy-Neutral Access Control

Principles. This multiplicity of security models contributed to an absence of agreement as to what should be the security model to implement in a distributed platform or operating system kernel, since each system supports his own classes of security policies based on a platform-specific security model. This motivated a new approach to authorization called *policy-neutral access control*, where authorization mechanisms should be agnostic vis-à-vis a security model. Since no single policy can claim to capture all the system protection requirements for different execution environments [37], the main advantage of policy-neutral authorization is to support multiple security models with a single mechanism. Thus, a wide range of models and policies can be enforced without need to change the security infrastructure, which is also more modular. In the case of wide-area networking, this approach allows to federate multiple security policies with a unified authorization mechanism. Indeed, federation of security policies is a key issue when implementing a secure infrastructure for global computing, where multiple heterogeneous networks, each governed by its own security policy, must be connected transparently.

Policy Description and Composition Languages. Several languages have been designed, both to describe authorization policies in a policy-neutral manner, or to reconcile different security policies. For instance: the ASL [31] authorization language supports SoD and Chinese Wall security models; the ORION++ [8] system enforces different policies in database environments. In each case, enforcement mechanisms are clearly separated from the security policy, a single security server supporting multiple authorization policies. Policy reconciliation languages such as [51] define policy algebras, abstract descriptions independent of enforcement mechanisms, which allow to decentralize policy specifications, and possibly to support unknown policies, expressed as incomplete specifications. Another approach is that of [46], where policy composition amounts to the composition of automata.

Some Implementations Some policy-neutral access control mechanisms have already been implemented for different types of kernels. For instance: the DTE-based confinement mechanisms for UNIX processes [3]; the support of multiple security models implemented in Linux as kernel modules [52]; or, the SELINUX architecture [48]. In the MACH micro-kernel [39], access request interception is realized by a policy-neutral security server inside the micro-kernel, while policy-decision is implemented outside the micro-kernel as an security server, which is dependent on a specific security model, and which can be replaced. Other implementations can be found in extensible [25] or component-based [32] kernels.

3 Selecting the Type of Kernel

The operating system provides applications with an abstract view of resources contained in a location: CPU, memory, file system, network bandwidth, etc. It offers primitives to manage resource sharing and protection. The system is generally composed of two types of components, which may, or may not reside within the same protection domain:

- The *kernel* has access to all hardware resources: there, the CPU can execute security-sensitive instructions in privileged mode.
- Higher-level *system services* are directly used by applications.

3.1 First Trends

General laws stating the impact of an OS architecture on security or performance are hazardous, since the drawn conclusions may be contradicted by the design of some specific systems. Nevertheless, in what follows, we try to sketch some general directions for trade-offs.

The security of the architecture strongly depends on whether services are located within the kernel protection domain, or are implemented as separate processes. Creating several distinct domains is in favor of stronger security: each application remains confined within its own address space, and cannot corrupt other parts of the system. However, this separation is made at the expense of system performance: each access request to hardware resources coming from an application-level domain requires a system call, costly due to multiple context switching with the kernel protection domain.

3.2 Different Types of Kernels

Monolithic Kernels. Traditional *“monolithic” kernels* like Linux or PALMOS integrate system services within the kernel protection domain. This solution results in a large-size and unstructured OS providing little flexibility. The isolation of a specific component within the kernel remains difficult. Upgrades often require recompiling the complete kernel. Due to its complexity, the OS contains many security loopholes: attacks on a kernel component are almost impossible to confine and will, more often than not, take control of the overall system. Despite such limitations, these systems generally present good performance results, since execution remains confined within the kernel address space, without need for context switching.

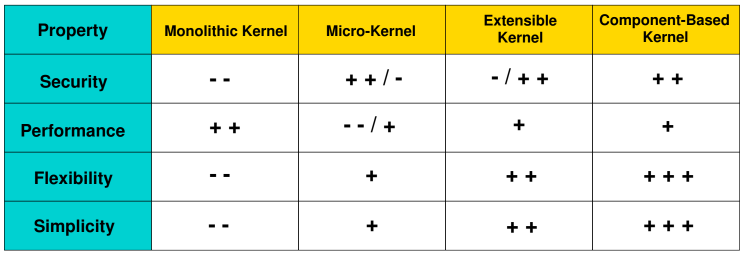
Micro-Kernels. In *micro-kernels* like QNX [29], services are run as separated processes which communicate via IPCs (Inter-Process Communication). Both the protection domains, which behave as confinement units, and the reduced kernel size make those systems much safer. They are also more flexible: the modular kernel structure allows to dynamically insert new services within the OS. Reduced performance may be the inherent limitation of this type of architecture, due to the cost of IPC mechanisms. Note that the overhead of an access control mechanism is lower when the authorization server is located outside the kernel, since the number of cross-domain invocations is then reduced. The trade-off of performance is in that case a weaker security.

Extensible Kernels. *Extensible kernels* [17, 20] are currently preferred for embedded systems, notably those with a single address space like SPIN [6]. Eliminating from the architecture unnecessary abstractions – like heavyweight processes, which are replaced by threads – improves performance, since cross-domain context switching is less costly. These minimal kernels only contain the absolutely essential services to multiplex hardware resources. Hence, the TCB is smaller and easier to certify, resulting in higher-assurance kernels. However, these systems are quite vulnerable to attacks without any additional protection mechanism [1], since they can be easily reconfigured by dynamically downloading modules within the kernel which may contain malicious code.

Component-Based Kernels. Finally, *component-based kernels* like THINK [18] are extensible kernels which allow a greater flexibility thanks to a more uniform architectural model : the entire kernel is built from an assembly of elementary units of reconfiguration called *components*, which are composable and reusable to design minimal dedicated systems. Compared to traditional systems, performance results show no significant degradation due to componentization.

[1] Extensible kernels are often written using strongly-typed programming languages: well-defined language semantics allows to produce safety proofs, which can be checked before downloading code into the kernel.

A Comparison. The above discussion is summarized in figure 1. Note that as the structure of the kernel is more clearly defined, its architecture offers more implementation hooks for flexibility.



Property	Monolithic Kernel	Micro-Kernel	Extensible Kernel	Component-Based Kernel
Security	- -	+ + / -	- / + +	+ +
Performance	+ +	- - / +	+	+
Flexibility	- -	+	+ +	+ + +
Simplicity	- -	+	+ +	+ + +

Fig. 1. Comparison of Different Types of Kernels.

4 Placing Security Mechanisms at the Right Abstraction Level

4.1 Hardware Solutions

Protection mechanisms may be implemented at the *hardware level*. Such a solution is truly minimal, enforces complete mediation, and provides fail-safe and tamper-resistant protection against internal, unintentional, or malicious threats to system integrity. Historical protection domains were implemented as hardware mechanisms in MULTICS [4]. Other examples of trusted hardware protection include: the MMU (Memory Management Unit) which confines applications by defining separate address spaces; co-processors with a secure mode such as [36]; or bus-level exclusion mechanisms [26].

4.2 Language-Based Solutions

Protection may also be *language-based*, using type-safe programming languages such as JAVA. These languages do not allow direct memory addressing, and enforce complete mediation using the concept of *interface*, unique access point to system resources. Language-based security solutions are quite flexible, and allow to realize easily fine-grained access control. Yet, their security is somewhat weaker than OS-level protection mechanisms, the attacker often taking complete control of a vulnerable run-time system. These solutions also require additional software layers (compilers, virtual machine) to be trusted. Thus, the TCB size is increased, making the system possibly more difficult to certify. For instance, see [30] for a comparison between language and OS-based security mechanisms.

4.3 Some Implementation Techniques

Between those two extremes co-exist a wide range of authorization mechanisms which can be implemented from kernel to application levels. A detailed comparison is beyond the scope of this report. Traditional solutions for discretionary access control like capability-based systems [35] are well-known. They only provide limited protection to confine the execution of programs with root privileges. Different forms of interposition allow enforcement of mandatory security policies: system call interception [5, 43], the system call level providing a well-defined interface to enforce complete mediation of security-sensitive operations; authorization hooks in kernel space [52]; and software wrappers to sandbox applications [1] or kernel modules [21]. Other solutions, more related to dynamic optimization, like program shepherding [33] or return address protection [15] can also be used to thwart buffer overflow or code injection attacks within the kernel.

The invasiveness of an access control mechanism will generally decrease as it is placed closer to the kernel, since context switching between user and kernel spaces will be less frequent. Reducing the distance – in terms of number of software layers to cross – between the authorization mechanism and the system resources to protect will also improve security, making mechanisms less likely to be bypassed, and introducing fewer security flaws. In that case, however, security enhancements will more difficult to reconfigure, requiring extensive changes or kernel recompilation.

5 Evaluating the Assurance Level

After selecting a security model and a type of kernel, and designing a security architecture where security mechanisms are placed in the most vulnerable software layers, one key issue remains: to convince other parties of the trustworthiness of the security membrane implementation. For low assurance levels, testing methodologies may be sufficient, using both unit testing and system testing to ensure that individual components are properly integrated. Kernels aiming for higher assurance levels would need to be verified using formal evaluation methodologies like the Common Criteria (CC) [40]. Code verification techniques needed to reach a given level of assurance may also be used to verify code safety for modules downloaded within the security membrane [41]. Existing verification algorithms and tools are currently limited to evaluating small systems, partly due to the computational power needed to carry out large-scale verification. Thus, proving correctness of complex security membranes is not yet within reach.

6 Towards Component-Based Security Membranes

In the previous sections, we explored some of the main trade-offs between security, flexibility, performance, simplicity, and trustworthiness, for different design parameters when implementing a security membrane. We now concentrate on two key properties for membranes in a global computing environment, often conflicting, and which need to be reconciled: reconfigurability and security. We then explore the benefits of a component-oriented design for implementing flexible yet secure membranes.

6.1 Reconciling Security and Reconfigurability

The advent of global computing has required infrastructures to become more adaptable and reconfigurable, to cope with the increasing number of wireless networks and the heterogeneity of mobile terminals. At stake is the extreme dynamicity of those environments, where both users and executable code are mobile. Execution contexts are characterized by the addition of new features on the fly, frequent downloads of platform updates, and personalization of existing services. However, reconfigurability puts security at risk, for instance when untrusted components are downloaded into a security infrastructure.

These infrastructures are generally extremely vulnerable and unreliable. Attacks, of increasing strengths and numbers, take advantage of the multiplicity of available paths for information flow, in overly complex systems. They also thrive on the size of extremely decentralized infrastructures, where applying a uniform security policy is nearly impossible.

Unfortunately, so far, reconfigurability has been considered orthogonally to protection. Among many explanations:

- Designs for security architectures ensure safe component reconfiguration within the limits of a specific security model. Reconfigurability has been applied with limited success to the security architecture itself that often remains monolithic.
- Security has simply been "forgotten" in a number of adaptable middleware for safety-critical and real-time systems.
- Traditional protection techniques are difficult to adapt to the context of wide-area systems. The architecture may be too monolithic, or demand too many computational and networking resources.

A compromise between security and flexibility has to be reached to win the trust of end users. Can secure yet reconfigurable membranes be designed and implemented using the same abstractions for reconfiguration and security architectures?

6.2 Components: a Solution for Reconfigurable and Secure Membranes?

We propose to reach such a goal by adopting a component-based approach to membrane design. This choice is motivated by a need for an integrated vision of security in global computing environments: indeed, grafting isolated and perimetric protection mechanisms on existing infrastructures is highly inadequate, since many loopholes and security breaches in the system are overlooked. A major advantage of component-based technology is to obtain with a single design a system that is simultaneously reconfigurable and secure.

Components are usually described as entities encapsulating code and data, endowed with an identity, which appear in software systems as units of execution, configuration, administration or mobility: “a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition to third parties” [49]. Building a system according to the component paradigm amounts to composing reusable units in a recursive way, since the assembly of a group of components allows creating new components at a higher level of abstraction.

Component technology enables to master the complexity of software infrastructures in terms of specification and implementation. The architect can choose the right level of abstraction to describe and observe the system, and, at this level, is given a homogeneous view of the infrastructure. The key argument in favor of Component-Based Software Engineering (CBSE) is to reduce the costs of software development, distribution, and maintenance, by reuse or mutualization of existing components. CBSE is well adapted to the dynamic needs of global computing environments, since it allows adapting and dynamically extending infrastructures, by addition, replacement, upgrade, or reconfiguration of components.

A component is not only a unit of reconfiguration. It is also a unit of protection. Although security properties are not in general compositional, CBSE enables to preserve security properties by including additional security mechanisms, for instance, by the use of type-safe languages to avoid any forms of interference between components during the operation of composition.

6.3 Benefits of Components for Security Membranes

Components appear as an promising solution to some the main design issues we discussed in this paper:

- Component-based design is security policy-neutral. Decoupling security policy decision components from the components containing the protection enforcement mechanisms enables to adopt a policy-neutral approach to security membrane design: hence, the same infrastructure mechanisms can be reused to enforce different authorization policies.
- CBSE also allows building a great variety of types of platforms, for instance in the domain of operating systems, ranging from monolithic and extensible systems to micro- and exo-kernels. We already saw in section 3 that in general the performance impact of componentization remains reasonable.
- Furthermore, by offering a fine-grained view of system functionalities, CBSE leaves a lot of freedom to the localization of protection mechanisms, which can be placed closer to the hardware, if stronger security is needed; or closer to applications, if simplicity and compatibility with existing software, e.g., no kernel recompilation, are the leading design requirements.
- Finally, thanks to its ability to realize truly minimal kernels, CBSE allows to build platforms with potentially high assurance levels.

These arguments are in favor of a component-based solution to achieve a good compromise between the various criteria we identified for the design of a security membrane. Given the breadth of the problem, much research remains still to be done in order to fully validate this claim.

7 Conclusion

The notion of security membrane appears as an emerging concept in the design of secure languages for global computing environments. Membranes separate the computational behavior of a site from the security code controlling access to site-located resources. However, an integrated view of the involved design issues is not yet achieved.

This brings forth new challenges for a practical implementation, which cannot be addressed by mere restriction to language-based aspects. In this paper, we studied some of these challenges, putting the focus on platform-related issues, in particular at the operating system level. We identified four critical design parameters for an effective implementation, namely the choice of a security model, the type of architecture for the execution environment, the layers at which to place security mechanisms, and the assurance level of the platform. In each case, we discussed possible trade-offs between key properties for a security membrane such as security, flexibility, performance, simplicity, and trustworthiness.

The proposed solution to reach a compromise between these properties is to design and implement a security membrane using the component-based paradigm, which maintains a delicate balance between reconfigurability and security. This choice enables the creation of different types of infrastructures without real performance penalties. The concern for modularity allows exploring several security models, and placing the access control mechanisms at different abstraction levels. Furthermore, component-based design may create truly minimal infrastructures, which could help to reach high assurance levels, to win the trust of end users. This approach allows adapting straightforwardly the system to the changes which may occur during its life-cycle, without really endangering the security of the infrastructure, since a component is both a unit of reconfiguration and of security. To further explore these ideas, more studies of integration of component-based concepts in practical implementations of security membranes will need to be considered. Thus, we hope this promising approach will emerge in the near future as a solution for the design of secure yet reconfigurable membranes.

References

1. A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings USENIX Security Symposium*, 2000.
2. S. Ames, M. Gasser, and R. Schell. Security Kernel Design and Implementation: An Introduction. *IEEE Computer*, 16(7):14–22, 1983.
3. L. Badger, D. Sterne, D. Sherman, K. Walker, and S. Haghinghat. Practical Domain and Type Enforcement for UNIX. In *Proceedings IEEE Symposium on Security and Privacy (S&P'95)*, 1995.
4. D. Bell and L. Lapadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, Electronics Systems Division, Bedford USAF Base, DoD, 1976.
5. M. Bernaschi, E. Gabrielli, and L. Mancini. REMUS: A Security-Enhanced Operating System. *Transactions of Information and System Security (TISSEC)*, 5(1):36–61, 2002.
6. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczyncki, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.
7. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A Logical Framework for Reasoning about Access Control Models. *ACM Transactions of Information and System Security*, 6(1):71–127, 2003.
8. E. Bertino, S. Jajodia, and P. Samarati. Supporting Multiple Access Control Policies in Database Systems. In *Proceedings IEEE Symposium on Security and Privacy (S&P'96)*, 1996.

9. K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, Bedford USAF Base, DoD, 1997.
10. W. Boebert and R. Yain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings National Computer Security Conference*, 1985.
11. G. Boudol. A Generic Membrane Model. In *Proceedings Workshop on Global Computing (GC'04)*, 2004.
12. D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings IEEE Symposium on Security and Privacy (S&P'89)*, 1989.
13. L. Cardelli, G. Ghelli, and A. Gordon. Types for the Ambient Calculus. *Information and Computation*, 177:160–194, 2002.
14. D. Clark and D. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings IEEE Symposium on Security and Privacy (S&P'87)*, 1987.
15. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks. In *Proceedings USENIX Security Symposium*, 1998.
16. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
17. D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.
18. J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : a Software Framework for Component-Based Operating System Kernels. In *Proceedings USENIX Annual Technical Conference (USENIX'02)*, 2002.
19. D. Ferraiolo, R. Sandhu, S. Gavrila, and D. Kuhn. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions of Information and System Security*, 4(3):224–274, 2001.
20. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'97)*, 1997.
21. T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings IEEE Symposium on Security and Privacy (S&P'99)*, 1999.
22. V. Gligor, S. Gavrila, and D. Ferraiolo. On the Formal Definition of Separation of Duty Policies and their Composition. In *Proceedings IEEE Symposium on Security and Privacy (S&P'98)*, 1998.
23. D. Gorla and R. Pugliese. Enforcing Security Policies via Types. In *Proceedings Conference on Security in Pervasive Computing (SPC'03)*, 2003.
24. D. Gorla and R. Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *Proceedings International Colloquium on Automata, Languages and Programming (ICALP'03)*, 2003.
25. R. Grimm and B. Bershad. Providing Policy-Neutral and Transparent Access Control in Extensible Systems. Technical Report UW-CSE-98-02-02, University of Washington, 2002.
26. T. Halfhill. ARM Dons Armor : TrustZone Security Extensions Strengthen ARMv6 Architecture. Microprocessor Report, 2003. Document availble at the URL: `http://www.arm.com/miscPDFs/4136.pdf`.
27. M. Harrison, W. Ruzzo, and J. Ullman. Protection in Operating Systems. *Communication of the ACM*, 19(8):461–471, 1976.
28. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
29. D. Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Micro-Kernels and other Kernel Architectures*, 1992.
30. T. Jaeger, J. Liedtke, and N. Islam. Operating System Protection for Fine-Grained Programs. In *Proceedings USENIX Security Symposium*, 1998.

31. S. Jajodia, P. Samarati, and V. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proceedings IEEE Symposium on Security and Privacy (S&P'97)*, 1997.
32. T. Jarboui, J.-P. Fassino, and M. Lacoste. Applying Components to Access Control Design : Towards a Framework for OS Kernels. In *Proceedings International Conference on Dependable Systems and Networks (DSN'04)*, 2004.
33. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings USENIX Security Symposium*, 2002.
34. B. Lampson. A Note of the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.
35. H. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
36. D. Lie, C. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'04)*, 2004.
37. P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, and J. Farell. The Inevitability of Failure: the Flawed Assumption of Security in Modern Computing Environments. In *Proceedings National Information Systems Security Conference*, 1998.
38. F. Martins and V. Vasconcelos. Controlling Security Policies in a Distributed Environment. Technical Report DI–FCUL–TR 04–01, Department of Computer Science, University of Lisbon, 2004.
39. S. Minear. Providing Policy Control over Object Operations in a Mach-Based System. In *Proceedings USENIX Security Symposium*, 1995.
40. National Institute Of Standards and Technology. Common Criteria for Information Technology Security Evaluation (CC). Technical Report CCIMB-99-031, 1999.
41. G. Necula. Proof-Carrying Code. In *Proceedings ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997.
42. S. Osborn, R. Sandhu, and Q. Munawer. Configuring Role Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and Systems Security*, 3(2), 2000.
43. N. Provos. Improving Host Security with System Call Policies. In *Proceedings USENIX Security Symposium*, 2003.
44. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
45. A. Schmitt and J.-B. Stefani. The M-calculus: a Higher-Order Distributed Process Calculus. In *Proceedings ACM Symposium on Principles of Programming Languages (POPL'03)*, 2003.
46. F. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
47. R. Simon and M. Zurko. Separation of Duty in Role-Based Environments. In *Proceedings IEEE Computer Security Foundations Workshop (CSFW'97)*, 1997.
48. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Ansersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings USENIX Security Symposium*, 1999.
49. C. Szyperski. *Component Software Systems*. Addison-Wesley, 2002.
50. U.S. Department of Defense. Trusted Computer Security Criteria (TCSEC). Technical report, 1985.
51. D. Wijesekera and S. Jajodia. Policy Algebras for Access Control - The Predicate Case. In *Proceedings ACM Conference on Computer and Communications Security (CCS'02)*, 2002.
52. C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings USENIX Security Symposium*, 2002.