Virtual Security Kernel: A Component-Based OS Architecture for Self-Protection

Ruan He   Marc Lacoste
Orange Labs
Security and Trusted Transactions Dept.
{ruan.he,marc.lacoste}@orange-ftgroup.com

Jean Leneutre
Telecom ParisTech
Network, Mobility and Security Dept.
jean.leneutre@telecom-paristech.com

***Abstract*—This paper presents VSK, a lightweight adaptable OS authorization architecture suitable for self-protection of pervasive devices. A "virtual" management plane, separate from execution resources, is defined for full run-time control by applications of their execution environment. This plane also performs non-invasive and yet effective authorization thanks to optimized access request checking. The VSK component-based architecture provides flexibility both in the execution plane (for resource customization) and in the management plane (for run-time reconfiguration of authorization policies). Policy-neutrality is achieved by adopting the attribute-based paradigm for access control enforcement. Evaluation results show that despite such flexibility, the overhead of this kernel architecture remains low.**

I. Introduction

The advent of pervasive computing has made devices become aware of their environment, adapting themselves to continuously evolving situations. For such context-aware systems, a secure OS design is one of the most important issues: mobility and openness induce multiple threats, as unknown, potentially malicious modules may be dynamically loaded into the operating system to upgrade or tune its functionalities, which may compromise the security of the device. Moreover, multiple security policies and models may be used for different environments. Therefore, a flexible security architecture with run-time reconfiguration capabilities is required to meet this diversity of security requirements.

However, the administration overhead of a security infrastructure usually remains high. One promising direction initiated by IBM is to extend context-awareness to the security mechanisms themselves to make them autonomic [1]. In this approach, protection schemes are automatically adapted at run-time according to the actual security requirements of the environment. This adaptation process is managed by an autonomic control loop in 4 steps: monitoring, context analysis, decision-making, and execution.

In [2], [3] we proposed a 2-level autonomic architecture for pervasive systems security: a cluster-level self-protection loop dynamically enforces security policies for a subnet, while a node-level loop realizes self-protection for each mobile device in the subnet. This paper focuses on the OS architecture enabling to realize the *execution* step at the node level to tune authorization mechanisms. Two main requirements should be satisfied:

1) Full customization with limited performance overhead: the execution environment should offer enough flexibility to tune running applications according to the environment, while meeting resource limitations of embedded devices.

2) Flexible security mechanisms: devices will move between multiple environments, each with its own security policy described in a specific security model. A policy-neutral security architecture enabling dynamic policy reconfiguration is thus needed to support those different models.

In this paper, we propose a new OS authorization architecture called *Virtual Security Kernel (VSK)* which provides strong and yet flexible security while still achieving good performance. VSK is a dynamic but lightweight management plane separate from execution resources which enables applications to fully control their execution environment at run-time. VSK includes non-invasive protection mechanisms thanks to optimized access control. Its component-based design allows flexibility both in the execution plane for resource customization and in the VSK plane for run-time reconfiguration of authorization policies – allowing to manage security mechanisms from an autonomic layer to make a whole subnet self-protected, as shown in [3]. Policy-neutrality is also achieved due to a clear separation of authorization attributes from rules thanks to attribute-based access control (ABAC) enforcement.

The remainder of this paper is organized as follows. After reviewing related work (Section II), we present the VSK architecture (Section III). We then describe a VSK implementation using the Think [4] component-based OS framework (Section IV), and present some evaluation results (Section V).

II. Related Work

The problem was tackled up to now either from the OS architecture, dynamic reconfiguration, or security standpoints.

OS architectures have considerably evolved in flexibility [5], [6] as the OS structure becomes more explicitly defined, and the kernel itself is getting smaller: system services are externalized from a core structure as modules, servers, libraries, or extensions which may specialized. Configuration manager architectures [7] and Service-Oriented Architectures (SOA) [8] went even further with no predefined kernel, enabling runtime control for each application. However, for embedded systems, component-based design is perhaps the most well-established paradigm [4], [9]: modularity is improved by encapsulating services into components. No specific OS structure is imposed, different system designs being possible by flexible arrangement of components. The system can also be reconfigured from design up to run-time. Those multiple benefits made this approach a natural setting for the VSK design.

Many *reconfiguration mechanisms* were also proposed to modify a system during its life-cycle. Build-time extensible systems like Linux remain limited to a modular architecture during the design phase. Library extensible systems [6] contain code to extend the kernel at run-time. Dynamic component image loading [10]–[12] enables to load a component image to the target system during execution. Finally, dynamic implementation loading [5], [13] allows replacing a component implementation without modifying the component state. Unfortunately, those systems did not go as far as fully-automated OS reconfiguration.

Finally, in terms of *security*, there is no real agreement on the right security model, e.g., access control lists [7], capabilities [11], multi-level security (MLS) [14], or Domain and Type Enforcement (DTE) [15]. More expressive models like RBAC [16], context-aware access control [17], [18], or Usage Control (UCON) [19] are not yet much used in an embedded setting. To overcome this diversity, several policy-neutral authorization architectures were proposed to make authorization mechanisms independent of a specific security model [20]–[26][1][2]. Some architectures supported policy reconfiguration, but only started to tackle the issue of self-protection [24] – autonomic security being up to now mainly discussed in terms of architecture [1], with few solutions for information systems only (e.g., [27]).

III. VSK Design

A. Architecture Overview

In a VSK design, the architecture of a self-protected device can be divided into 3 layers, as shown in Figure 1. The *execution space* is the execution environment of components. The *Virtual Security Kernel (VSK)* controls the execution space, for instance to enforce authorizations on shared resources. At the top, an *autonomic manager* coordinates and enables automatic adaptation of management strategies enforced in the VSK according to context information – in our case to reconfigure VSK access control policies[3]. In this paper, we focus on the first two layers, self-protection mechanisms based on VSK being described in [3].

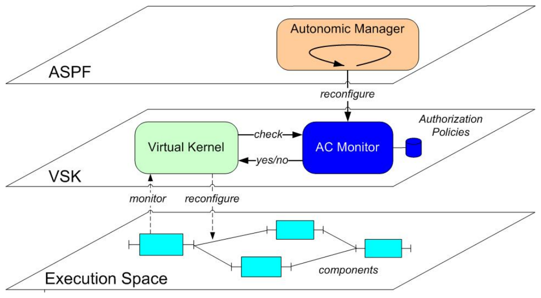


Figure 1. A 3-Level Autonomic OS Architecture.

[1]Singularity [23] achieves efficient software isolation using type-safe languages, instead of a lightweight reference monitor as in VSK. Authorization is based on extended notions of ACLs allowing to express a large range of authorization models, but remains discretionary, unlike VSK mandatory access control. Despite flexible relationships between applications, Singularity does not allow dynamic reconfiguration of authorization policies.

[2]The Qubes [26] secure hypervisor presents some similarities with VSK by moving most of the resource management out of the kernel, resulting in a thin OS running over bare metal. In Qubes, a form of security management plane is split between system VMs and the administrative domain (Dom0). However, Qubes mainly tackles isolation, while VSK addresses access control. In Qubes resources are virtualized, whereas simply abstracted as components in VSK. Qubes presents some level of reconfigurability due to dynamic VM spawning, but not the full adaptation capabilities both in and out of the kernel made possible by the VSK component-based design. Finally, Qubes does not address self-protection issues.

[3]The upper plane is realized by a dedicated framework called ASPF (*Autonomic Security Policy Framework*) [3]. Strategies for autonomic security adaptations are currently hardcoded, using if-then-else rules. A more generic approach based on Domain-Specific Languages (DSLs) is currently under investigation to specify richer types of adaptation policies.

1) Design Approach: VSK is an improvement of the exo-kernel paradigm in the setting of component-based systems. On the one hand, a component-based OS abstracts application and OS services as components, viewed as units of design and of execution. This approach provides a homogeneous view of the different system entities and enables their dynamic reconfiguration. On the other hand, exo-kernel architectures demonstrated the feasibility to export out of the kernel most OS services as applications, only thread management and protection mechanisms remaining inside the kernel. VSK goes one step beyond: those services become "virtual" in their behavior, i.e., kernel entities will not be involved any more during most of the execution of applications. This design, while still guaranteeing effective protection, yields significant performance improvements, as shown by our implementation.

2) An Example: Consider a mobile terminal connected either to a domestic, private, secure network or to an outdoor, public, insecure network. When the terminal is part of the domestic network, an OS service such as a file system may access not only public files (e.g., user documents) but also sensitive files (e.g., system security settings). When the terminal joins the public network, new drivers may need to be installed on the terminal to upgrade its communication capabilities if the radio access technology was previously unknown. These drivers may request access both to sensitive and public files and may represent a risk for system security which should be mitigated dynamically, taking into account the ambient risk level. VSK allows to reach that goal by providing support for self-protection, moreover reducing administration overhead.

The first time the OS service tries to access a sensitive file, it asks the VSK for authorization. If the VSK grants the request, it creates a secure binding to the file. Hence, at the next invocation, the service can directly access the file without any control (Figure 2).

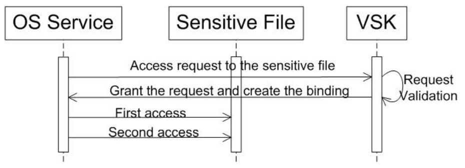

OS Service
Sensitive File
VSK
Access request to the sensitive file
Request Validation
Grant the request and create the binding
First access
Second access

Figure 2. VSK Access Validation.

3) VSK Architecture: A system built according to a VSK design consists in two parts (see Figure 3): a runtime environment for components (*execution space*) and a control plane (*VSK*) to supervise their execution in an almost invisible manner. This separation not only provides application-specific customization for executing programs but also protects the kernel from attacks by malicious applications. Furthermore, a simple but effective access control mechanism in VSK guarantees the security of sensitive resources in the execution space. VSK consists of a *Virtual Kernel (VK)* and an *Access Control Monitor (ACM)*.

The VK plays the role of a kernel when needed (e.g., during the initialization or reconfiguration of the execution space) and remains hidden in the background when no more changes occur. It catches events from the execution space, decides whether a reconfiguration of that space is required, and reacts accordingly. The VK may also dynamically reconfigure itself according to high-level policies, reacting to decisions taken in the autonomic layer.

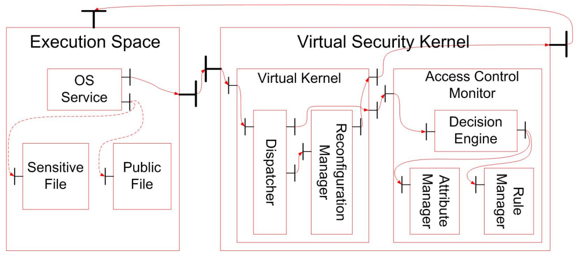


Figure 3. VSK Architecture.

The ACM enforces access control on resources in the execution space, based on authorization attributes and rules which may be dynamically loaded, modified, or replaced.

B. The Virtual Kernel

In the execution space, functional code is encapsulated as *components*, inter-component communication being realized via *bindings*. The VK has three main functions: (1) a lightweight control "only when needed" over components in the execution space; (2) control mechanisms to reconfigure the execution space; and (3) access control enforcement during binding creation. This design allows to meet performance requirements, since the kernel overhead is minimized during execution of applications by limiting interactions between the execution space and the VSK. From the customization viewpoint, applications can dynamically choose their needed services which increases implementation freedom. Finally, this architecture guarantees strong protection by strict control over creation or modification of bindings.

The VK consists of a *dispatcher* and a *reconfiguration manager*. The dispatcher provides an effective and extensible event-based communication mechanism between threads. For instance, the execution space, VSK, and a communication module for interacting with the external environment may work in parallel. The dispatcher also manages concurrent access to resources by enforcing mutual exclusion. Finally, this mechanism enables to add easily new security modules in the ACM or extend kernel services simply by defining new event types.

The reconfiguration manager provides run-time component and binding management capabilities to modify the execution space based on decisions made in the kernel. For instance, it creates bindings when access requests are granted by the ACM. It may also add new components in the execution space. In the case of permission revocation, it removes all corresponding bindings. Above all, this component is a key building block for efficient access control enforcement. In DEIMOS [7] or sHype [25], access controls were optimized by a one-time only enforcement until the next change of authorization policy. In the VK, a similar mechanism is proposed, access control only taking place during the creation of bindings. Unlike previous systems, all bindings are managed at run-time, and created only when needed for use. Once access to a resource is granted the first time and the corresponding binding created by the reconfiguration manager, subsequent access requests to the resource are not checked any more.

C. The Access Control Monitor

The ACM enforces policy-neutral resource access control: this component is basically a flexible reference monitor where different security policies may be selected dynamically.

In our architecture, active components (*subjects*) try to access passive components (*objects*). SELinux has shown its flexibility in separating policy from enforcement mechanisms, with a clear distinction between abstract security identifiers (SIDs) and dedicated security modules which can process those SIDs. Our ACM goes one step further in flexibility by separating *security attributes* from *rules*, following the attribute-based vision of access control (ABAC) [28]. One SID may be assigned different security attributes (e.g., role, type, security level, domain, etc.), some of which can be used to compute permissions. When a subject requests access to an object, the ACM first gets the subject and object security attributes, computes permissions via these attributes, and makes the access decision.

This separation improves the access control flexibility, different security policies being supported with a common meta-model (ABAC). It also helps capturing the diversity of security requirements in a mobile context: a device moving to another environment may only require selecting new security attributes. Thanks to the VSK component-based design, the change of security attributes or rules can be simply realized by replacing components.

1) Attribute-Based Access Control: The ACM contains three sub-components: a *Decision Engine* for decision-making, and an *Attribute Manager (AM)* and a *Rule Manager (RM)* for manipulating attributes and rules respectively. As shown in Figure 4, to request access to an object, a subject first issues a request `getPermission(subject,object,operation)` to the RM via the Decision Engine. The RM then asks the AM for the security attributes of the corresponding entities. For example, in the case of an RBAC policy, the needed attribute for the subject would be its role – a subject-role table being maintained in the AM RBAC sub-component. Once the role is retrieved, the RM can then compute the permissions assigned to this role for the requested operation on the object using the policy rules relevant for the role.

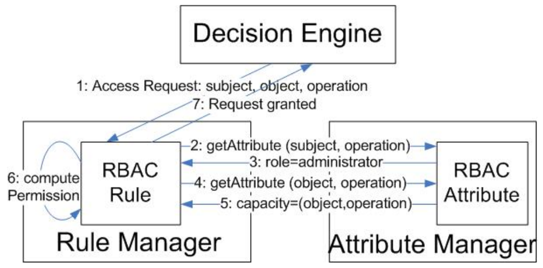

Decision Engine
1: Access Request: subject, object, operation
7: Request granted
RBAC Rule
6: compute Permission
2: getAttribute (subject, operation)
3: role=administrator
4: getAttribute (object, operation)
5: capacity=(object,operation)
RBAC Attribute
Rule Manager
Attribute Manager

Figure 4. ACM Request Processing in the RBAC Model.

As the RM-AM interface is security model-dependent, a generic access control description language should be chosen to allow enforcement of a large number of different authorization policies. This language should offer a uniform representation of policies for many authorization models, so that there may be no difference between changing the policy in the same model and switching from one model to another. In our system, we extend ASL [29] to take into account separation of security attributes from rules. Model-dependent authorization can be represented by the predicate:

```
md-grant(subject, object, operation, model) :-
  attribute(s_attribute, subject, operation, model),
  attribute(o_attribute, object, operation, model),
  rule(permission, s_attribute, o_attribute, model).
```

For the RBAC security model, it may be interpreted as:

```
md-grant(subject, object, operation, RBAC) :-
  attribute(role, subject, operation, RBAC),
  attribute((object, operation), object, operation, RBAC),
  rule(permission, role, (object, operation), RBAC).
```

The predicate value is then easily derived from an ASL RBAC policy specification.

2) Permission Revocation: When modifying dynamically access control policies, to avoid privilege abuse the system should revoke outdated authorizations. This task is generally complex since decision-making may be decentralized throughout the whole system. With VSK, revocation management is simpler: as all access authorizations are enforced by the reconfiguration manager, once there is a change of policy, that manager will remove all relevant bindings it created in the execution space to guarantee authorization policy consistency. When new access requests are issued for which some authorizations were revoked, the VSK will rebuild the adequate bindings based on the updated security policy.

IV. Implementation

VSK is specified using the Fractal component model and implemented using the Think component-based OS framework. Fractal [30] is a hierarchical and reflective component model to design, implement, deploy, and manage software systems. A Fractal *component* is both a design-time and a run-time entity, acting as unit of encapsulation, composition, and configuration[4]. Think [4] is a C implementation of the Fractal model. Using Think, an OS architect can build a system from components without being forced into a predefined kernel design. This flexibility made it quite easy to implement the different components of the VSK architecture. The Think reconfiguration framework [12] was notably helpful to implement dynamic policy reconfiguration operations. Note that the Think compiler is actually a cross-compiler enabling to compile and debug an OS kernel on one platform, and install it on another platform: VSK was implemented and evaluated on a 2.7GHz Dell OptiPlex 740 desktop PC with Linux/Ubuntu 9.04 and 1GB of RAM, but may be easily ported to other platforms like ARM or AVR thanks to the Think cross-compilation toolchain. The VSK implementation is further described in appendix.

V. Evaluation

We evaluate the VSK architecture in terms of performance in Section V-A. We then provide a qualitative evaluation of the security of our system in Section V-B.

A. Performance

We made the simplifying assumption that a typical embedded OS was able to run at most 10 threads (*subjects*) which could invoke about 60 system calls (*objects*). To compare on an even basis the different security models, we computed permissions for such a number of subjects and objects for `read` and `write` operations. Authorization policies for 7 basic security models (ACLs, capabilities, DTE, MLS, RBAC, UCON, ORBAC) and 2 hybrid models (UCON/ORBAC and Lipner [31] combining confidentiality- and integrity-oriented MLS) were specified using the ABAC extension of ASL, and enforced in the Think implementation of the VSK.

1) Access Control Monitor Overhead: The performance of the ACM component was first evaluated alone. In the ACM, access control checking is performed in 4 steps: (1) get subject and object attributes; (2) optionally retrieve context information (if the security model is context-aware); (3) select access rules and compute permissions; and (4) manage sessions (if the model has a session control function).

To assess the authorization overhead, we measured the overall time in micro-seconds necessary to perform steps 1-4 for the 9 considered models. The results shown in Figure 5a are the average of a series of 1000 subject/object access requests sent to the ACM [5].

DTE policies group subjects and objects into domains and types: permission lookup for a given (domain, type) pair thus requires to go through the whole rule table. MLS policies need both subject and object security levels to perform a clearance comparison, resulting in a similar overhead. RBAC uses only roles to manage user privileges, which reduces the attribute fetching time. However, permission computing times are higher, since navigation through all object-operation pairs is necessary to retrieve the permission corresponding to a given role. UCON controls authorization either before, during, or after the resource usage, which implies an additional session management overhead. ORBAC integrates context information into authorization rules to capture privilege dependency according to organizations and/or situations, which induces a small overhead to retrieve context information.

Combined security models are also explored to specify policies which could not be described within only a single model – e.g., the Lipner MLS model combining confidentiality and integrity labels, and a UCON-ORBAC model taking into account usage control and session management.

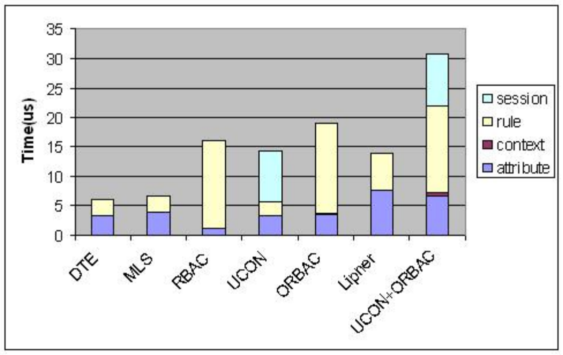
Overall, authorization overheads are bigger for combined models than for basic models, since two validations instead of one are needed. All overheads remain however much smaller than those of conventional ACL- or capability-based models thanks to the separation of access attributes from rules in authorization enforcement.

2) Full VSK Kernel Overhead: The VSK architecture also contains the VK component which manages user/kernel mode switching and reconfiguration of the execution space: when a resource access request is issued from the execution space, the VK dispatcher handles concurrency issues and calls the ACM to compute permissions. If the request is granted, the VK reconfiguration manager then creates the binding to access the resource.

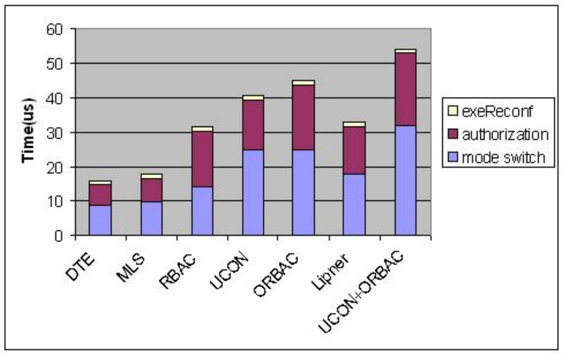
Figure 5b shows the total VSK kernel overhead, i.e., including both VK and ACM components. The average times spent in the kernel are shown in micro-seconds for 1000 access requests sent to the VSK from the execution space for different security models. Mode switching, authorization, and reconfiguration times are also shown. The mode switching cost somewhat depends on the

[4]Components provide *server interfaces* as access points to the services that they implement, while functional requirements are expressed by *client interfaces*. Components interact through *bindings* between client and server interfaces. Any component may have *attributes* that represent primitive properties. The Fractal model also defines standard interfaces to control the internal structure of a component at run-time, for instance for binding management or adding/removing sub-components. The software architecture of a system is then given by its hierarchy of bound components.

[5]The overheads for ACL- and capability-based models do not appear in the figure, and are respectively 1273μs and 1271μs. Such large figures may come from the fact that those models directly use subject or object identities for authorization, resulting in a large subject-object-operation table being maintained in the ACM.



(a)



(b)

Figure 5. VSK Kernel Overheads: (a) ACM Component Only; (b) Full Kernel (VK+ACM Components).

complexity of the security model: for UCON, additional session management information should be maintained; similarly, ORBAC requires context information retrieval processing. Results tend to show that the cost of execution space reconfiguration is negligible compared to mode switching or authorization overheads.

3) Comparison with Micro-Kernel Design: We compare the efficiency of our kernel design with other OS architectures such as micro-kernel or exokernel. The micro-kernel philosophy maintains only core OS functionalities in the kernel, moving less essential services and applications to userland – exokernels may be viewed as an extreme application of this principle, since only access control and concurrency management remain in the kernel.

As shown in Figure 6, we thus measured the invocation cost in micro-seconds for VSK and micro-kernel architecture prototypes, both implemented using the Think OS framework. The duration of a raw invocation without protection is also mentioned, as a comparison basis with a pure component-based architecture. Two VSK prototoypes implementing DTE and UCON policies were chosen to show the influence of the security model on the results.

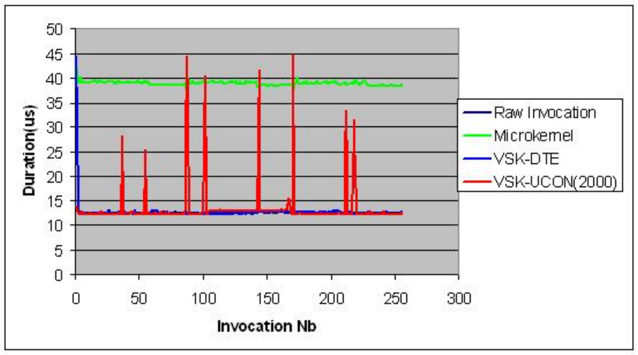


Figure 6. VSK vs. Micro-Kernel .

A raw invocation without security takes 12.5μs, and rises to 39μs for the micro-kernel. With the VSK, this time sharply decreases after the first invocation. In a micro-kernel, authorization hooks are statically bound into a reference monitor to intercept each access request, inducing high access control overheads. VSK applies a "one-time only" access checking mechanism, i.e., after the first check, there is no more control until the next change of authorization policy. Indeed, as of the second invocation, VSK becomes more efficient than the micro-kernel [6].

The main difference between VSK and the micro-kernel architecture is also run-time support for system reconfiguration. Overall, the performance gain might be explained by a more lightweight, and more dynamic system architecture, bringing additional benefits like dynamic authorization policy reconfiguration or permission revocation management.

Afterwards, the VSK performance becomes comparable to the raw invocation time. This last result also demonstrates that VSK offers the high performance of component-based OS architectures, with no additional cost to provide strong security.

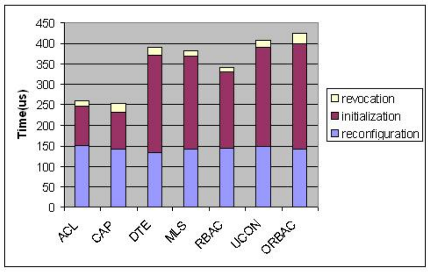
4) Reconfiguration Overhead: We compute the reconfiguration overhead for changing attributes (e.g., assignment of a new role), rules, or security models. The results in micro-seconds of each type of reconfiguration are shown in Figure 7 for the 7 basic security models (averaged over 1000 runs). Such operations are performed in 3 steps: (1) reconfigure the security components in the kernel, e.g., replace an access attribute component by another; (2) initialize the reconfigured components; and (3) revoke outdated access decisions by removing some existing bindings.

In the attribute case, the component reconfiguration overhead is similar for each model, but the initialization phase depends on the complexity of the model. For rules, the most costly models are RBAC (to build the role-permission relation) and ORBAC (due to multiple organization/context management). A model reconfiguration involves both change of attributes and rules. Overall, security model reconfigurations appear quite time-consuming, and should be restricted only for situations when switching between two very heterogeneous environments. Attribute or rule reconfigurations are however much lighter (only a few times the cost of an invocation), and perfectly feasible in practice.

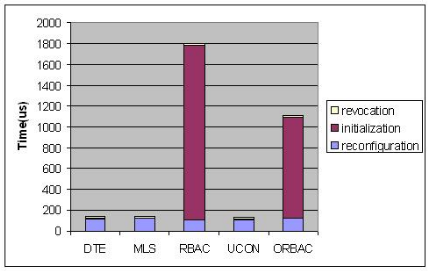
5) Kernel Occupation Rate: The principle of the "virtual" security kernel is to remain hidden during most of the execution time. To assess the efficiency of the VSK from that perspective, we use the *Kernel Occupation Rate (KOR)* metric, defined as the ratio of kernel running time over total execution time.

Figure 8 shows the KOR evolution when invoking repeatedly a given method both for micro-kernel and VSK prototypes. For

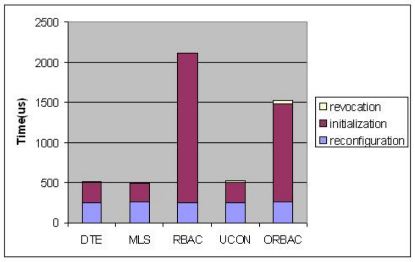
[6]For UCON, authorizations performed throughout the system life-cycle appear as periodic peaks, but the average overhead still remains low.



(a)



(b)



(c)

Figure 7. Reconfiguration Overheads: (a) Attribute Reconfiguration; (b) Rule Reconfiguration; (c) Security Model Reconfiguration.

the micro-kernel, the times spent in user land and in the kernel are respectively 12.5 μs and 39.5μs, yielding a KOR of 68.4%. For VSK, the KOR decreases with the life-time T of component bindings, as authorization checks are performed only at binding creation. The KOR is under 10% for $T > 1ms$ and under 2% for $T > 10ms$. Typical security policy updates occur at most every minute (to allow physical switching between environments) yielding an effective KOR of less than 1%, making the VSK much less intrusive than the micro-kernel.

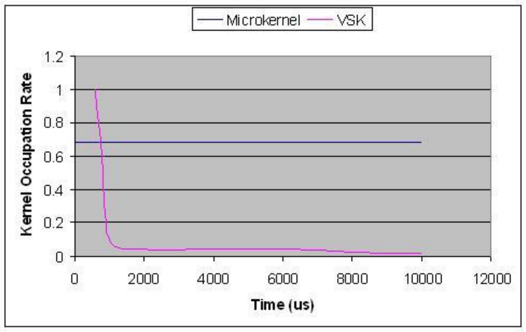


Figure 8. Kernel Occupation Rate: VSK vs. Micro-Kernel.

B. Security Analysis

The security of the VSK relies on the following assumptions: (1) the VSK is isolated from the execution space using a MMU-like hardware mechanism to avoid circumventing the ACM; this mechanism also prevents bypass of VSK security checks in the execution space; and (2) the Autonomic Manager and the VSK interact via a secure channel, to avoid rogue third parties to directly update node authorization policies inside the VSK.

We argue that the VSK architecture has all the distinguishing features of a security kernel – or minimal implementation in an OS of the security-relevant features that mediates all accesses, is protected from modification, and is verifiable as correct. Indeed, our VSK intercepts all access requests (*completeness*), and cannot be modified from the execution space (*isolation*). Moreover, its simple architecture should facilitate proof of correctness (*verifiability*). The VSK also provides additional features like: support of multiple security models (*flexibility*), dynamic choice of the most adequate security configuration (*manageability*), and easy introduction of new security models in the kernel (*extensibility*). Hence, the VSK enables strong and yet flexible protection for applications running in the execution space during their whole life-cycle – from design, deployment, execution, maintenance, to un-installation.

VI. Conclusion

This paper presented the VSK component-based OS authorization architecture which provides strong and yet flexible security while still achieving good performance, making it applicable to make pervasive devices self-protected. The definition of a dynamic but lightweight management plane separate from execution resources allows applications to control and customize their execution environment at run-time, yielding a highly adaptable OS architecture. Including protection mechanisms in this plane also reduces the authorization overhead without compromising overall security, thanks to one-time checks only during creation of bindings until the next change of authorization policy. The component-based structure of the VSK control plane allows making the authorization architecture policy-neutral to support multiple security models, but also to reconfigure at run-time kernel access control modules, yielding a flexible and dynamic security architecture. A clear separation of authorization attributes from rules thanks to the ABAC paradigm also improves access control granularity.

Directions for future work include hardware mechanisms to guarantee VSK integrity and non-circumventability. Regarding self-protection aspects, we are currently working [3] on autonomic policy specification languages for self-protection extending the Tune [32] approach to describe wrapping and adaptation of managed elements, such as the interface and life-cycle for reconfiguring protection in the VSK.

References

[1] D. Chess, C. Palmer, and S. White, "Security in an Autonomic Computing Environment," *IBM Systems Journal*, vol. 42, no. 1, pp. 107–118, 2003.

[2] R. He and M. Lacoste, "Applying Component-Based Design to Self-Protection of Ubiquitous Systems," in *ACM Workshop on Software Engineering for Pervasive Services (SEPS)*, 2008.

[3] R. He, M. Lacoste, and J. Leneutre, "A Policy Management Framework for Self-Protection of Pervasive Systems," in *International Conference on Autonomic and Autonomous Systems (ICAS)*, 2010.

[4] M. Anne et al., "Think: View-Based Support of Non-Functional Properties in Embedded Systems," in *International Conference on Embedded Software and Systems (ICESS)*, 2009.

[5] O. Krieger et al., "K42: Building a Complete Operating System," *Operating Systems Review*, vol. 40, no. 4, pp. 133–146, 2006.

[6] D. Engler, M. Kaashoek, and J. O'Toole, "Exokernel: an Operating System Architecture for Application-Level Resource Management," in *ACM Symposium on Operating System Principles (SOSP)*, 1995, pp. 251–266.

[7] M. Clarke and G. Coulson, "An Architecture for Dynamically Extensible Operating Systems," in *International Conference on Configurable Distributed Systems (ICCDS)*, 1998.

[8] N. Milanovic and M. Malek, "Service-Oriented Operating System: A Key Element in Improving Service Availability," in *International Service Availability Symposium (ISAS)*, 2007.

[9] "eCos," `http://ecos.sourceware.org/`.

[10] I. Lee, "DYMOS: A Dynamic Modification System," Ph.D. dissertation, University of Wisconsin, 1983.

[11] B. Bershad et al., "Extensibility, Safety and Performance in the SPIN Operating System," in *ACM Symposium on Operating Sytems Principles (SOSP)*, 1995.

[12] J. Polakovic and J.-B. Stefani, "Architecting Reconfigurable Component-Based Operating Systems," *Journal of System Architecture*, vol. 54, no. 6, pp. 562–575, 2008.

[13] J. Helander and A. Forin, "MMLite: A Highly Componentized System Architecture," in *ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, 1998.

[14] D. Bell and L. La Padula, "Secure Computer System: Unified Exposition and Multics Interpretation," MITRE Corporation, Bedford, MA, Tech. Rep. MTR-2997, 1975.

[15] L. Badger, D. Sterne, D. Sherman, K. Walker, and S. Haghighat, "Practical Domain and Type Enforcement for UNIX," in *IEEE Symposium on Security and Privacy*, 1995.

[16] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.

[17] M. Damiani, E. Bertino, B. Catania, and P. Perlasca, "GEO-RBAC: A Spatially Aware RBAC," *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, no. 1, 2007.

[18] A. Abou El Kalam et al., "Organization Based Access Control," in *IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2003.

[19] J. Park and R. Sandhu, "The $UCON_{ABC}$ Usage Control Model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128–174, 2004.

[20] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *USENIX Security Symposium*, 2002.

[21] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *USENIX Annual Technical Conference*, 2001.

[22] R. Grimm and B. Bershad, "Providing Policy-Neutral and Transparent Access Control in Extensible Systems," University of Washington, Tech. Rep. UW-CSE-98-02-02, 1998.

[23] T. Wobber, A. Yumerefendi, M. Abadi, A. Birrell, and D. Simon, "Authorizing Applications in Singularity," in *ACM EUROSYS Conference*, 2007.

[24] M. Lacoste, T. Jarboui, and R. He, "A Component-Based Policy-Neutral Architecture for Kernel-Level Access Control," *Annals of Telecommunications*, vol. 64, no. 1-2, pp. 121–146, 2009.

[25] R. Sailer et al., "Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor," in *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[26] J. Rutkowska and R. Wojtczuk, "The Qubes OS Architecture," Invisible Things Lab, Tech. Rep., 2010.

[27] B. Claudel, N. De Palma, R. Lachaize, and D. Hagimont, "Self-Protection for Distributed Component-Based Applications," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2007.

[28] E. Damiani, S. Di Vimercati, and P. Samarati, "New Paradigms for Access Control in Open Environments," in *International Symposium on Signal Processing and Information*, 2005.

[29] S. Jajodia, P. Samarati, and V. Subrahmanian, "A Logical Language for Expressing Authorizations," in *IEEE Symposium on Security and Privacy*, 1997, pp. 31–42.

[30] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and its Support in Java," *Software: Practice and Experience*, vol. 36, pp. 1257–1284, 2006.

[31] S. Lipner, "Non-Discretionary Controls for Commercial Applications," in *IEEE Symposium on Security and Privacy*, 1982.

[32] O. Chebaro, L. Broto, J.-P. Bahsoun, and D. Hagimont, "Self-TUNe-ing of a J2EE Clustered Application," in *IEEE Conference on Engineering of Autonomic and Autonomous Systems*, 2009.

Appendix

A. VSK Implementation Overview

Figure 9 shows the implementation architecture of a VSK-based system. The main components are: the *bootloader* that starts the VSK kernel and launches applications; the *execution space* where application-level components may be installed; and the *VSK* which is the kernel proper.

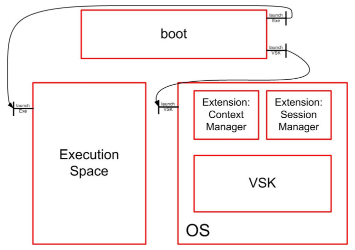


Figure 9. VSK-Based System Overview.

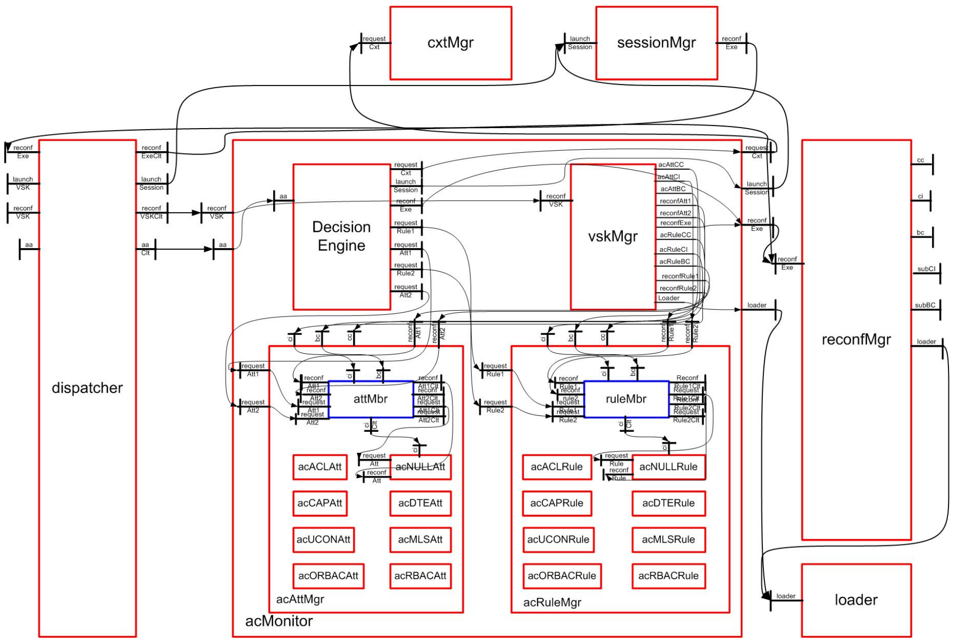


Figure 10. Detailed Structure of the VSK.

The latter contains a *dispatcher* that supervises the execution space and transmits requests to different modules in the VSK, an *access control monitor* for authorization decision-making, a *reconfiguration manager* which reconfigures the execution space based on decisions taken by the ACM, and a *component loader* to dynamically load new components into the system. Some *extension modules* also allow to extend kernel functionalities. A policy management framework called *ASPF (Autonomic Security Policy Framework)* is also implemented above the VSK to manage authorization policies in an autonomic manner [3].

B. The Execution Space

The execution space is a run-time environment for application-level components. Two components may interact when bindings are created between their respective interfaces, access controls being performed transparently by the VSK.

Our architecture fully takes advantage of the Fractal component model, where the run-time behaviors of components may be supervised by a *membrane* through the definition of multiple specialized controllers. The membrane non-functional logic is realized by a specific component which may perform different reconfiguration tasks on the execution space such as dynamic component loading, binding and component life-cycle management, and access request mediation from the execution space to the VSK[7].

C. The VSK Kernel

The main purpose of the VSK is to retrieve access requests from the execution space, validate them, and provision access to resources. Figure 10 presents its main components.

1) The Dispatcher: When access to a resource is requested, the dispatcher changes from user to kernel mode, aggregates similar requests then transmitted to the ACM for validation. If access is granted, the reconfiguration manager is then invoked to create the requested bindings in the execution space.

2) The Access Control Monitor: The ACM contains several components. The *Attribute Manager* allows to fetch access attributes. As different security models may be supported in the OS, a model-independent external interface is provided as a facade towards model-dependent attributes implemented as sub-components. The *Rule Manager* allows to select the access control rules which match given attribute values. The *Decision Engine* makes the access decision based on the attributes and rules. Finally, the *Kernel Manager* may dynamically reconfigure the ACM internal structure to load and install security policies expressed in different models, enabling highly reconfigurable kernel authorization enforcement, contrary to conventional OSes.

3) The Reconfiguration Manager: This component creates the right bindings in the execution space once a request has been validated by the ACM. It also destroys the adequate bindings between application-level components in case of permission revocation.

D. Kernel Extensions

Although VSK is designed to be policy-neutral, some security models may require additional kernel functionalities. For example, in the ORBAC model, authorization is context-aware. A context manager component may thus be added in the kernel to handle context data, possibly in cooperation with an external context management infrastructure. Similarly, for the UCON model where authorization is session-based, a specific session manager component may be inserted.

[7]The `BindingController` interface allows to create/remove bindings at run-time. The `ContentController` interface may be used to install in the execution space a composite component including multiple sub-components. The `LifeCycleController` interface allows to start or stop application-level components. The `AttributeController` interface allows to query / change attribute values.