A Survey of Security Research for Operating Systems*

Masaki HASHIMOTO†

Abstract

In recent years, information systems have become the social infrastructure, so that their security must be improved urgently. In this paper, the results of the survey of virtualization technologies, operating system verification technologies, and access control technologies are introduced, in association with the design requirements of the reference monitor introduced in the Anderson report. Furthermore, the prospects and challenges for each of technologies are shown.

1 Introduction

In recent years, information systems have become the social infrastructure, so that improving their security has been an important issue to the public. Besides each of security incidents has much more impact on our social life than before, the number of security incident is increasing every year because of the complexity of information systems for their wide application and the explosive growth of the number of nodes connected to the Internet.

Since it is necessary for enhancing information security to take drastic measures concerning technologies, managements, legislations, and ethics, large numbers of researches are conducted throughout the world. Especially focusing on technologies, a wide variety of researches is carried out for cryptography, intrusion detection, authentication, forensics, and so forth, on the assumption that their working basis is safe and sound. The basis is what is called an operating system in brief and various security enhancements running on it are completely useless if it is vulnerable and unsafe. Additionally, if the operating system is safe, it is an urgent issue what type of security enhancements should be provided from it to upper layers.

Therefore, there exist many researches and implementations for a long time, about improving security of operating systems themselves, and security enhancements to provide from operating systems as safety area, including the TCSEC (Trusted Computer System Evaluation Criteria) from the US-DoD, Trusted Solaris, SELinux, and so on. However,

†Graduate School of Information Security, Institute of Information Security

*This article is originally authored in Japanese by Masaki Hashimoto, Ruo Ando, Toshiyuki Maeda, and Hidehiko Tanaka, published in IPSJ Transactions on Advanced Computing Systems (ACS), Vol.5, No.2, pp.51-62, Mar. 2012, revised and translated by Masaki Hashimoto in Nov. 2013.

many researches are on-going today because the former researches are now the essential elements to reduce a lot of threats but insufficient to support our healthy information society.

In this paper, some recent researches are introduced letting operating systems be the most basic software to guarantee information security, especially focusing on virtualization technologies of operating systems, verification technologies for operating system programs, and access control technologies. The organization of this paper is as follows. First, in the chapter 2, classification policy about security researches in this paper are showed, and the relation between three technologies mentioned above are explained in association with the design requirements of the reference monitor in TCSEC. Then, some researches about each technologies and remaining issues are presented in the chapter 3, 4, and 5. Finally, the chapter 6 summarizes this paper as conclusion.

2 Security Research for Operating Systems

Trusted Computing Base (TCB) is the basic component to construct trusted computer systems, and the concept is defined in the TCSEC (Trusted Computer System Evaluation Criteria)[1] in 1985, involving some components such as a reference validation mechanism that is called reference monitor, a formal security policy model which is enforce by reference monitor, and so on. The Anderson report[2] listed the three design requirements that must be met by a reference validation mechanism; i) the reference validation mechanism must be tamper proof, ii) the reference validation mechanism must always be invoked, iii) the reference validation mechanism must be small enough to be subject to analysis and tests, the completeness of which can be assured.

Operating systems are met the requirement ii) by nature because they are the undermost layer of software stacks of information systems. Therefore, many researches have been taken place so far to enhance information security making a good use of operating systems as a reference monitor. Additionally, operating systems researches from the other perspective have been carried out for meeting the requirements caused by evolutions of computer hardware and social requests because operating systems have a role of the execution environment for application programs in its origin. Especially in recent years, various researches are on-going all over the world to meet the newly requirements from embedded systems, mobile systems, and cloud computing.

In this paper, those researches are surveyed in terms of information security, namely, this paper classifies recent researches into following three areas, relating to attacks against operating systems themselves and against programs running on operating systems. In brief, all the researches in this paper are explained as countermeasures for these attacks; operating system verification technologies for the former attacks, access control technologies for the latter attacks, and virtualization technologies for the both attacks.

1. Virtualization Technology

Virtualization technologies have been studied for a long time to utilize hardware resources effectively and wide spreading globally as a basic infrastructure for cloud computing of late. Virtualization technologies are related to all the requirements of reference monitor.

2. OS Verification Technology

   OS verification technologies are a kind of formal verification proving the properties of OS code, like safeness, reliability, validity, and so on. Virtualization technologies and OS verification technologies are related to the requirement (iii) , since a reference validation mechanism must be analyzed, tested, and ensured to be complete.

3. Access Control Technology

   Access control technologies are provided from a reference monitor and ensuring the safeness of whole information system, related to the requirement (ii). Access control technologies rely on the assumption of reference monitor's safeness and managing security of upper layer programs.

3 Virtualization Technology

In recent years, significant improvements in hardware performance have made research and development of OS virtualization technologies an active area of research. OS virtualization technologies have also been applied as foundational technologies for cloud computing, and strengthening security in such applications is an important issue. This section describes virtualization technology research over the past few years, organized according to four aspects: virtual machine observation via hypervisors, virtualization of main memory devices, virtualization of I/O devices, and verification of the completeness of a virtual machine. Future issues in these areas are also discussed.

As to the reasoning for dividing things into these four categories, first, proper implementation of information security measures requires that events occurring within the information system are viewable by the defensive side, but not the attacking side. In view of the active research being applied to this end, hypervisor monitoring of the virtual machine are taken up in the first section. Next, as the current objects of hypervisor monitoring are predominantly memory and I/O devices, this section is also organized according to those topics. Finally, exclusion of software that would allow an attacker to perform monitoring requires that all execution sequences, including boot processes, be subjected to completeness verification. Related research assumes that hardware support is taking place in numerous locations, and this subject is treated in its own section.

3.1 Technologies for Virtual Machine Monitoring by Hypervisors

A hypervisor is situated between the virtual machine and hardware, and performs resource management and scheduling for the virtual machine. One of the benefits gained by using a hypervisor is that on the operation side multiple virtual machines can be managed simultaneously. This allows resources to be assigned or reclaimed dynamically, according to the operational situation. A second benefit is a strengthening of virtual machine security, in that use of a hypervisor tightens access control over general-purpose OSs, which may include vulnerabilities due to increased complexity. Hypervisors also allow the construction of observation and analysis devices that cannot be detected by malware. Representative examples of hypervisor implementation are given in Xen[3], KVM[4], and VMware[5].

The concept of using hypervisors to allow monitoring, observation, defense, and isolation of virtual machines is known as Virtual Machine Inspection (VMI) [6]. VMI is considered an effective means of detecting and preventing unauthorized access, due predominantly to three characteristics: the virtual machine is unable to alter code on the hypervisor side, the hypervisor is able to monitor all aspects of the virtual machine, and code issued from the virtual machine can be captured. Many papers relevant to this study have been published regarding virtual machine observation.

Methods of VMI observation include both active and passive methods. In active methods, the status of the virtual machine is externally obtained at fixed intervals. Volatility [7] is an example of research in this area, and uses such extracted information to obtain and analyze a snapshot of memory. Passive methods are methods in which events within the virtual machine, such as resource access, trigger extraction of related information. Representative examples of such research are Lares[8], Xenprobes[9], and VMScope[10].

There are two approaches to implementing both active and passive methods: in-the-box and out-of-the-box methods. There have been numerous discussions of the possibility of attacker detection of defender-side observational devices and of the potential for elimination of semantic gaps[2][11], as well as the tradeoff costs of implementing those. At present it is considered difficult for observational devices to be detected from within the virtual machine, so out-of-the-box methods are generally employed. There is one significant limitation to such methods, however, in that the amount of observable information is less than with in-the-box methods.

3.2 Virtualization of Main Memory Devices

Handling of main memory devices, paging mechanisms in particular, is an important issue when implementing virtualization technologies. The physical addresses used by the virtual machine are artificial addresses virtualized by the hypervisor. Coordinating access to the true physical addresses therefore generally requires twice the number of

[2]Semantic gaps refer to an inability to reconstruct semantic information regarding VM-internal resource access from information received by the hypervisor. In other words, eliminating the semantic gap would mean making it possible for the hypervisor to reconstruct semantics related to resource access on the virtual machine.

address conversions as in the non-virtualized case. This doubled paging, peculiar to virtualization technologies, can be performed either in hardware or in software.

A representative example of virtualizing main memory devices in software is the Shadow Paging technique of Xen. Shadow Paging catches main memory page faults to virtualize main memory accesses by the virtual machine. A mechanism called the Shadow Page Table is used to convert between the physical addresses recognized by the virtual machine and the physical addresses on the actual machine. There is ongoing research regarding how to use Shadow Page Table modifications to detect rootkits and other malware that uses kernel extensions.

For example Panorama [13] proposes a method for detecting contamination; it uses Google Desktop[14] as a case study. Ether[15] also proposes a method for detecting malware using hardware-based virtualization assistance features. Similarly, Ether[16] proposes a method for detecting malware operating externally from within the guest OS; this method is through acquisition of RDTSC data.

From the attacker side of things, in 2006 SubVirt[17] and Blue Pill[18] were proposed as ways of taking advantage of the powerful isolation methods a hypervisor should provide; this would create a rootkit that cannot be detected from the guest OS. Similarly, Ristenpart et al. [19] showed that it was possible to perform a side-channel attack on Amazon EC2 between virtual machines operating on the same physical machine. Chen et al. [20] performed a comprehensive investigation and classification of methods by which analysis by a hypervisor on the defender side could detect malware.

Representative examples of virtualizing main memory structures in hardware are Intel VT-d and AMD-V. In these, virtualization technologies can utilize hardware-based memory management, decreasing the load on the hypervisor resulting from address conversions and implementation. In Intel VT-d this is performed using the Extended Page Table, and an example of using this to detect unauthorized kernel extensions is HUKO [21]. Under HUKO the table that is shared in Shadow Paging is physically isolated and shared by each guest OS, which allows for eliminating the overhead of guest OS paging while still strengthening access control. SIM [12] is another example of using isolation to protect the address space used by the virtualization technology, but while SIM uses Shadow Paging, HUKO uses hardware-supported paging to reduce the load of VMM transitions and TLB load.

3.3 I/O Unit Virtualization

When security is tightened through measures such as access control from the hypervisor to the VM and integrity verification, it is necessary to implement I/O virtualization in addition to virtualization of the main storage devices, as introduced in the previous section. As in the case of the main storage, I/O virtualization can be implemented in either software or hardware.

In software I/O virtualization, I/O requests from device drivers are captured and interrupted. Split kernel drivers in Xen are a representative example, where memory shared between the VM and the hypervisor is allocated and I/O is virtualized using an event channel. Also, as a practical application of this technique, in XenAccess[22] the split kernel driver blktap is used to modify VM events and to reduce the load on the entire system. Furthermore, sHype[23] can enforce security policies when generating and deleting VM snapshots and generating virtual interrupts.

Another implementation is BitVisor[24], which is a hypervisor using the so-called para-pass-through architecture. In comparison to other hypervisors, BitVisor captures only those I/O requests from the VMt that are related to access control and encryption and enforces security through intermediate processing of these requests. Using this architecture, the hypervisor has to process only control I/O and data I/O, and it is unnecessary to implement protection and scheduling between VMs. BitVisor consists of about 20000 lines of code (20 KLOC) in its core section, and para-pass-through drivers are about 1/10th of ordinary drivers in terms of lines of code. Furthermore, from the perspective of the CIA triad of information security, in contrast to other hypervisors, which concentrate on ensuring confidentiality, BitVisor also includes effective functions for ensuring integrity in addition to confidentiality.

IOMMU supporting direct access to the hardware for VMs has been proposed for hardware I/O virtualization. IOMMU implements address remapping in hardware during direct memory access (DMA), allowing VMs to directly manipulate addresses of physical devices. In relation to this, Intel VT-d and TXT (Trusted Execution Technology)[25] can stop malicious code from being transferred via DMA. TXT intervenes into the process of address remapping during DMA and rejects DMA to the protected memory region. Furthermore, Intel VT-I and Intel VT-d can completely separate I/O spaces, including memory access of the guest OS, thus ensuring confidentiality.

3.4 VM Integrity Verification

With the increased incidence of attacks on VMs, research on integrity verification of structural elements of VMs has grown. For physical machines, devices such as TPM (Trusted Platform Module) are used for integrity verification, and a virtualized version of TPM (vTPM[26]) has been proposed for structural verification of multiple VMs running on top of a physical machine.

vTPM implements a split device driver in both the VM and the VM monitor in order to capture I/O requests from the VM to the TPM. There are two types of TPM-based trusted boot, namely SRTM (Static Root of Trust Management) and DRTM (Dynamic Root of Trust Management). SRTM is used in research by Sailer et al. [27] as well as in Bitlocker[28]. In their 2004 study, Sailer et al. proposed a method for maintaining a chain of trust all the way from the bootloader through the kernel to the application. In this way, the integrity of code and data loaded into the OS is automatically verified by TPM. Furthermore, in 2007, Kauer presented an attack method for SRTM and proposed

Open Secure Loader (OSLO), which uses the AMD SKINIT technology[29].

A more advanced integrity verification technique is HIMA[30], which is capable of verifying the consistency of TOCTTOU (time of check to time of use). In HIMA, priority is given to design objectives such as robust isolation and protection against TOCTTOU attacks. Toward that end, it uses techniques such as dynamic monitoring and memory protection for guest OSs.

HyperSentry[31] is another research project that focuses on integrity verification for the hypervisor. It uses out-of-channel dedicated monitoring structures and communication paths that cannot be detected by the verification target for verifying the integrity of the hypervisor itself. Specifically, a monitoring structure is established within the hypervisor, which communicates with a remote verifier via an SMI (System Management Interrupt) handler and IPMI (Intelligent Platform Management Interface). In this way, sufficient isolation is maintained from the hypervisor, and the integrity of the entire system, including the hypervisor, can be verified.

3.5 Prospects and Challenges

Problems related to OS virtualization technology to be addressed in future research include the application of all research results introduced above, as well as the further strengthening of the various security features related to virtualization. The gradual upscaling and increasing complexity of general-purpose OSs increases the risk of vulnerabilities. Since general-purpose OSs are expected to find various applications in the future, including in embedded products, the construction and isolation of environments implementing security features must be performed in the hypervisor, which is closer to the hardware.

For example, systems such as HUKO, which detect and stop incorrect kernel extensions, are under active development at various institutions for the purpose of strengthening security features related to access control. Hypervisors for the detection and analysis of malware are also being investigated, and methods for their implementation are currently sought after. Research is also being conducted on the detection of malicious operations inside VMs using hypervisors by eliminating the semantic gap[32].

4 OS Verification Technology

The OS kernel is the software base of computational systems, and its reliability and stability have a strong effect on the security of the entire system. For this reason, research on direct OS kernel verification has been conducted for many years. In this section, three verification methods are introduced; theorem provers, source code model checking and safe programming languages. These verification approaches are compared in Table 1. In this section, problems for future research on OS kernel verification are also presented.

表 1: Comparison of OS verification approaches

Verification method	Verification performance	Verification cost
Theorem provers (Section 4.1)	Excellent	Moderate
Source code model checking (Section 4.2)	Good	Good
Safe programming languages (Section 4.3)	Moderate	Excellent

4.1 Verification using Theorem Prover

A theorem prover is a program that takes the proof of a given theorem as input and determines its correctness [33, 34]. First, the safety and reliability of the OS kernel are expressed as a theorem, which is subsequently proven, and the reliability and stability of the OS kernel are examined and guaranteed by verifying the correctness of this proof by using a theorem prover. Specifically, for example, the OS kernel behavior can be expressed in terms of state transitions of an abstract state machine. By proving that this machine satisfies certain conditions (such as not allowing illegal memory operations, unexpected halting or infinite loops), the OS kernel properties can be verified and guaranteed.

One advantage of theorem provers is that they can verify any arbitrary properties that the assistant can take as input. However, one disadvantage is that the verification cost is exceedingly high since the proof must be constructed manually.

Kit [35] was the first OS kernel whose program was directly verified. Kit is an extremely small kernel consisting of approximately 300 lines of machine code (in machine language for an abstract von Neumann machine). Inside the kernel, task isolation is proven using the Boyer-Moore theorem prover [36]. Specifically, the abstract specifications and machine-language implementation that the kernel should satisfy are defined using the Boyer-Moore logic, and the Boyer-Moore theorem prover is used to prove that the isolation of each task is guaranteed by the abstract specifications as well as that the implementation of the abstract specifications is correct.

An example of a somewhat larger OS is seL4 [37], which is a kernel implemented in about 8700 lines of C code and about 600 lines of assembler code. This implementation realizes a formal specification, which has been guaranteed and verified using Isabelle/HOL [33] (it should be noted that since seL4 is a microkernel, it does not include complex parts such as memory management, which are accordingly not verified). The actual verification procedure was as follows. First, the properties and behavior that the kernel must exhibit are defined in terms of an abstract specification (for example, interfaces and behavior of system calls). Next, the executable specification is defined as an abstract implementation of the abstract specification. Particularly, a program implementing the abstract specification is written using a subset of the Haskell programming language [38], and the executable specification is automatically generated from its source code. The execution specification thus obtained is proven to correspond precisely to the abstract specification. Next, the execution specification is implemented using a subset of the C programming language. The semantics of that subset is formally defined in Isabelle/HOL, and the C-language implementation can be handled more or less directly by Isabelle/HOL. Finally, the C-language implementation is proven to correspond precisely to the execution specification. This proof required a total of about 22 man-years to complete.

4.2 Verification Based on Source Code Model Checking

Source code model checking [39, 40] applies model checking techniques [41] directly to program source code. More specifically, a model is extracted from source code given as input, and the program properties are verified and guaranteed by performing an exhaustive search of the states assumed by the program.

The merit of OS kernel verification using source code model checking is that it hardly involves any manual intervention, in contrast to the theorem prover method. However, one disadvantage is that model checking often requires considerable computational resources (CPU time and memory).

SDV (Static Driver Verifier) [42] is a framework for verification of Windows device drivers. Specifically, the source code model checker SLAM [43] is used to verify whether device drivers obey specifications denoted using the SLIC specification language (mainly the use of the kernel API). This is not intended for verifying the OS kernel itself. SDV is included into the Windows Driver Kit developed by Microsoft, and is already at a level where it can be used in deployment environments.

Furthermore, although not relying on direct source code model checking, a similar method is used in an attempt for OS kernel verification of the Nucleus part of Verve [44]. Verve is an OS with verified and guaranteed type safety. The Nucleus part of Verve is responsible mainly for memory management (garbage collection), thread management (stack management) and hardware management (such as interrupt processing and device drivers). Nucleus was verified using the following procedure. First, the assembly code implementation of Nucleus and the specifications met by Nucleus were represented using the Boogie [45] program verifier and passed as input to Boogie. Next, Boogie constructed verification conditions showing whether Nucleus behaved according to the specifications. The Z3 [46] SMT solver was used to verify whether these conditions were satisfied. Thus, the verification of Nucleus was essentially automatic. Nucleus consists of about 4500 lines of Boogie code (corresponding to about 1400 lines of assembly code). However, about 7% of that corresponds to manual annotations inserted as hints for the complete automation of the verification procedure. Furthermore, the representation of the assembly code implementation and specification in Boogie required about 9 man-months, and the automatic verification of Nucleus by Boogie required 272 s on a machine with a 2.4-GHz Intel Core2 CPU and 4 GB of RAM.

4.3 Verification using Safe Programming Languages

In this paper, the term “safe programming language” refers to a programming language which uses strict type checking to guarantee and verify that flaws such as illegal memory operations and code execution do not arise at program runtime. Representing an OS using a safe programming language allows strict type checking to guarantee and verify that the OS does not perform illegal memory operations or exhibit other undesired behavior. During program analysis, strict type checking classifies variables and functions according to their type (types represent integers and memory references, data and code, etc.), and checks if the program behaves according to those types at runtime. In this way, programs that have passed strict type checking are guaranteed against flaws such as illegal memory operations or code execution.

The merit of representing an OS using a safe programming language is that OS verification can be performed automatically through type checking in the programming language. Furthermore, compared to source code model verification, the computational resources (CPU time and memory) required for verification are smaller. However, a disadvantage of this method is that verification is limited to basic properties. Another disadvantage is that OS representation using a safe programming language imposes a great load on OS developers.

SPIN [47] is an OS microkernel supporting safe extensibility. In traditional microkernels, kernel safety is guaranteed by executing extensions with privileges different from those of the kernel itself. However, this has been associated with an overhead on inter-extension and extension-kernel communication. In contrast, SPIN kernel extensions are represented using the safe programming language Modula-3 [48], which guarantees safety even if they are executed with kernel privileges, thus reducing the communication overhead. Specifically, tasks requiring low latency, such as virtual memory management and network communication, have been implemented as kernel extensions using Modula-3. In SPIN, verification does not target the kernel itself. Furthermore, the kernel safety can be compromised by bugs in the Modula-3 compiler. Therefore, the Modula-3 compiler should be included into the trusted computing base (TCB) of the system.

Singularity [49] is an OS represented using assembly language and the type-safe programming languages C# [50] and Sing#, which is an extension of C#. Strictly speaking, Sing# is an extension of Spec#, which in turn extends C# by explicitly representing preconditions and postconditions for object methods. Sing# extends Spec# further by introducing inter-process communication channels. In Singularity, OS components are implemented as processes, and inter-component communication is conducted via the abovementioned channels. Specifically, Singularity consists of about 280000 lines of C# code and 90000 lines of Sing# code (or architecture-dependent assembly code), and type safety as well as inter-component independence are guaranteed for the parts represented in C# or Sing#. However, the parts responsible for memory and thread management are outside the verification scope. Since programs written in C# or Sing# are converted into type-safe assembly programs (a typed assembly language [51]) and type safety can be examined at the level of the assembly programs, in contrast to SPIN, it is unnecessary to include the C# and Sing# compilers into the system TCB.

TOS [52] is an OS kernel represented in the typed assembly language TALK [53, 54], which has been extended for the purpose of OS kernel representation. TALK allows for OS kernel representation and type verification, which has been difficult to perform using traditional typed assembly languages, by using type systems for handling variable-length arrays for memory representation, integer constraints for calculation of addresses as well as safe strong updates for memory (memory operations changing the type of the memory region). TOS is a limited-capability OS kernel consisting of about 3000 lines of TALK code. Its characteristic feature is that it uses TALK to represent the parts responsible for memory and thread management, which have not been targeted directly in OS verification research using other safe programming languages. This allows for memory safety to be verified and guaranteed using TALK type verification.

4.4 Prospects and Challenges

As mentioned above, direct OS verification has already been realized at the level of research, and future work will involve the application of verification techniques to more large-scale and practical OS s. It is conceivable that the several verification techniques introduced in the preceding section can be combined for that purpose. In fact, in Verve [44], the parts other than Nucleus are represented separately in C# (in other words, in a safe programming language), allowing for type safety of the entire OS to be guaranteed.

Furthermore, a more academic and technical problem is how to verify the OS safety in the increasingly popular multi-core CPU environments, which allow for multiple programs to be executed simultaneously [54]. Traditional research on OS verification usually assumes a single-CPU computing environment. Specifically, it is necessary to consider race conditions and problems with the memory coherence model [55, 56, 57] arising when multiple programs operate at the same time on shared memory.

5 Access Control Technology

Access control is one of the most basic and elemental techniques for maintaining information security, and it is also related to the preservation of confidentiality, integrity and availability. For this reason, as mentioned in Section 2, access control functionality is provided on top of the OS's TCB. Information security has been researched continually for close to half a century, and the first results of such research can be found in TCSEC[1]. Furthermore, in recent years, a highly abstracted access control system has been standardized in ISO/IEC10181-3[58]. In the following sections, security policy models, security policy description languages, security policy verification techniques, and access control mechanisms are introduced as the main structural elements of OS-based access control functions. And research trends are also shown in all of these areas, as well as problems for future research.

5.1 Security Policy Models

Information security requires different levels of confidentiality, integrity and availability depending on the use, and therefore various security policy models have been researched in the past to reflect this. The main types of policies target confidentiality preservation, integrity preservation and a combination of the two. Representative security policy models of each type include the Bell-LaPadula [59] (confidentiality), the Biba Integrity [60] and Clark-Wilson [61] (integrity) and the Chinese Wall [62] and role-based access control (RBAC) [63, 64] (combined confidentiality and integrity). Representative examples of recent research on security policy models are presented below.

The RBAC model [63, 64] is a role-based security policy model. Since access control within an information system can be easily matched to the responsibilities of people and groups in the real world, it is intuitively understandable and, if managed correctly, can be used to realize access control following the principle of minimal privilege. RBAC has been adopted in SELinux and various OSs, such as Solaris. Results of recent research on the RBAC model include its formal definition and an extension of the basic model [65] as well as its standardization and adoption into the ANSI standard[66]. In addition, administrative RBAC (ARBAC)[67, 68] has been developed and an RBAC model analysis[69] focusing on management problems has been conducted regarding the question of how and by whom the various relations in the RBAC system should be set. A method for the formal verification of the security features of RBAC implementations on actual OSs has also been proposed[70].

The task-based authorization control (TBAC) model [71] is a security policy model where multiple authorizations are grouped into an authorization procedure (authorization step, AS), and the authorization approval is determined by specifying the access subject, the accessed object, the details of the intended operation as well as the AS name and the process name within the AS. In TBAC, access rights assigned to subjects in each AS are activated or deactivated successively depending on the context. For example, fine-grained access rights can be specified in accordance with the transaction processing status. In this way, TBAC allows for an unobstructed view of the entire policy by structuring the authorization procedure in an abstract manner and allowing it to be used repeatedly as a subroutine.

5.2 Security Policy Description Languages

Security policy description languages are intended for representation of security policies and can be regarded as languages that describe access control specifications. In recent years, security policy description languages have been researched with the application of knowledge on mathematical logic, including description logic, with emphasis on problems including descriptive power with regard to the delegation, restriction and revocation of rights, grammatical clarity and readability for humans, semantic brevity and clarity, efficiency of the rights approval procedure and extensibility of the language. Below, representative examples of research on security policy description languages are introduced.

SecPAL[72] is a security policy description language for flexible description of access rights delegation between areas under decentralized management, under the assumption that the security policy of a relatively large-scale system is constructed as a module for each management area. SecPAL is a high-level security policy description language based on a constraint programming language, where the requirements for rights approval are met upon a successful query with respect to a cluster of nodes. SecPAL's grammar is close to that of a natural language, and meaning is constructed based on three inference rules. Through support for negative queries, recursive predicates, specifying the number of times that rights can be delegated and various other constraints, the language can represent a vast number of security policy models in a generic way. SecPAL remains the subject of active research as it has been implemented as a security policy description language for the cloud OS Windows Azure.

Lithium[73] is a security policy description language which can correctly infer logical negation in a formal manner. Lithium is a high-level language based on first-order predicate logic that supports inferences including negation by restricting recursive representation. Lithium is useful, for example, when merging multiple policies and analyzing the resulting access control specifications since it can provide formal verification that access has not been authorized. However, it also suffers from the problem that the delegation of access rights, which is required in many security policy models, cannot be represented simply. Since Lithium can be used without knowledge of its grammar and first-order predicate logic, a frontend supporting policy description using ordinary English is being developed[74].

5.3 Security Policy Verification

Security policy verification is a technique for checking whether the security policy obeys given access control specifications. Information systems are becoming increasingly large and complex, bringing about a dramatic increase in the amount of descriptions of security policies using languages such as those introduced in the preceding section, and in many cases the realized access policy specifications become incomprehensible to humans. To address this problem, a verification method allowing for described security policies to be easily analyzed is sought after and actively researched. Below, representative examples of research on security policy verification are introduced.

For ARBAC models, RBAC-PAT[75] is capable of verifying parameters such as the reachability and availability of information flow when a given role is taken as a point of origin as well as the compartmentalization and weakest point of an information system. RBAC-PAT simplifies the analysis of policies in ARBAC models where multiple administrators can change the access rights at any time. Specifically, it can verify the reachability and availability of a given role for a user, the relation between inter-role connotation and the smallest set. RBAC-PAT can also discover unnecessary roles and verify the information flow between objects.

PALMS[76] can verify the total information flow depending on the multi-level security (MLS) policy under analysis by defining a formal specification of the policy. Furthermore, it can automatically verify the conformance between two MLS policies. PALMS is implemented as a tool based on Prolog and can verify whether a given MLS policy obeys a ∗policy or simple security condition, as well as whether the MSL policy of an application is compatible with the MLS policy of the host OS. PALMS targets MLS policies only, but since in many cases actual information systems use multiple security policy models simultaneously, the question of how to enable verification in cases where MLS policies are compatible with other security policy models is currently being researched.

5.4 Access Control Mechanisms

Access control mechanisms are schemes for enforcing individual access control rules (described in a security policy description language) on information systems. Specific implementations such as the ACL and capability formats have been developed[77]. While the ACL format has been used for a long time, the capability format is still being researched since it has the merit of realizing access control based on the principle of minimal privilege. Furthermore, the integration of compulsory access control structures into existing OSs is being researched with an emphasis on compatibility with security policies. Below, representative examples of research on access control mechanisms are shown.

seL4[37] is a research-purpose OS based on an L4 microkernel with extended security. It implements a combination of the take-grant model [78] and the capability format, and imposes access control on all kernel objects. L4 kernel objects are threads, the address space, interprocess communication and untyped memory, which represents unused physical memory, where access restrictions for these objects are imposed and removed using the capability format. Another example of research on the capability format is Capsicum[79], which is an OS for research purposes that extends the standard UNIX API. It uses the capability format for easy and fine-grained sandboxing of applications and processes. Research is also under way on a design concept of an OS where Capsicum can easily separate individual applications rather than controlling the separation between the OS and the application layer in a unified manner.

The FLASK security architecture (FLSA)[80] consists of an object manager that enforces the execution of decisions based on a security policy, and a security server that provides decisions about granting and refusing access in accordance to a given security policy. These two structures communicate following a predefined protocol and serve as a reference monitor for the entire FLSA. One of the main features of FLSA is its ability to realize various security policy models in a flexible manner. FLSA has been implemented as SELinux in Linux and as SEBSD . The TrustedBSD MAC framework[81] has been proposed as similar enforced access control structure, which has been implemented in FreeBSD and Darwin.

5.5 Prospects and Challenges

Research on access control techniques has been conducted for a long time, and the fundamental theory has already been established. At the same time, extensions and applications of that theory following the changes in demand for information security are constantly sought after. The research introduced in the preceding section is progressing as information systems continue to form the base of society.

Future research should focus on access control specifications for cloud environments based on virtualization and for embedded systems such as smartphones, as well as the application of the above-mentioned achievements to environments with different assumptions required for realizing these specifications. Security policy models and description languages oriented toward distributed systems as well as distributed access control mechanisms for enforcing these models are particularly sought after, and relevant research is already being conducted at various institutions. For example, there is research on extending the application scope of FLSA from access control mechanisms for objects captured at the OS layer of standalone systems to mechanisms for objects captured at the application layer and in other system objects[82, 83].

6 Conclusion

This paper describes a result of survey of security research for operating systems and shows the prospects and challenges of this research area. According with the requirements of reference monitor, each of researches is classified into three research areas; virtualization technologies, os verification technologies, and access control technologies. Then, outlines and research examples are detailed respectively as well as problems. Specifically, VM monitoring by hypervisor, virtualization of main memory devices, I/O unit virtualization, and VM integrity verification are explained for virtualization technologies. And verification using theorem prover, source code model checking, and safe programming language are explained for OS verification technologies. Besides, security policy model, security policy description language, security policy verification, and access control mechanism are explained for access control technologies.

For the healthy development of our information society of the future, almost all the applications need security support by robust and dependable OS essentially, though many challenges are left to be solved for each of the elemental technologies, as introduced in this paper. It is believed that basic principles of OS have not changed for nearly half a century, because social requests for OS have not changed over the years. However, at the near future, they will vary greatly than ever according with the rapid spread of new computing environment, and therefore, the remaining challenges of various technologies discussed in this paper is expected to be resolved and utilized as a matter of course in the real world.

参考文献

[1] US Department of Defense: Trusted Computer Systems Evaluation Criteria, Technical Report CSC-STD-001-83, DoD Computer Security Center, Fort Meade, MD (1983).

[2] Anderson, J. P.: Computer Security Technology Planning Study, Technical Report Vols. I and II, USAF Electronic Systems Div., Bedford, Mass (1972).

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, New York, NY, USA, ACM, pp. 164–177 (2003).

[4] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux virtual machine monitor, *Proceedings of the Linux Symposium*, Vol. 1, pp. 225–230 (2007).

[5] Waldspurger, C. A.: Memory resource management in VMware ESX server, *SIGOPS Oper. Syst. Rev.*, Vol. 36, pp. 181–194 (2002).

[6] Garfinkel, T. and Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection, *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 1, ISOC, pp. 253–285 (2003).

[7] Volatile Systems, L.: The Volatility Framework: Volatile memory artifact extraction utility framework (2006).
https://www.volatilesystems.com/default/volatility.

[8] Payne, B. D., Carbone, M., Sharif, M. and Lee, W.: Lares: An Architecture for Secure Active Monitoring Using Virtualization, *Security and Privacy, IEEE Symposium on*, Vol. 0, pp. 233–247 (2008).

[9] Quynh, N. A. and Suzaki, K.: Xenprobes, a lightweight user-space probing framework for Xen virtual machine, *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 2:1–2:14 (2007).

[10] Breslau, L., Chase, C., Duffield, N., Fenner, B., Mao, Y. and Sen, S.: VMScope: a virtual multicast VPN performance monitor, *Proceedings of the 2006 SIGCOMM workshop on Internet network management*, INM '06, New York, NY, USA, ACM, pp. 59–64 (2006).

[11] Chen, P. M. and Noble, B. D.: When Virtual Is Better Than Real, *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Washington, DC, USA, IEEE Computer Society, pp. 133– (2001).

[12] Sharif, M. I., Lee, W., Cui, W. and Lanzi, A.: Secure in-VM monitoring using hardware virtualization, *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, New York, NY, USA, ACM, pp. 477–487 (2009).

[13] Yin, H., Song, D., Egele, M., Kruegel, C. and Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis, *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, New York, NY, USA, ACM, pp. 116–127 (2007).

[14] inc., G.: Inside Google Desktop.
http://googledesktop.blogspot.com.

[15] Dinaburg, A., Royal, P., Sharif, M. and Lee, W.: Ether: malware analysis via hardware virtualization extensions, *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, New York, NY, USA, ACM, pp. 51–62 (2008).

[16] Pék, G., Bencsáth, B. and Buttyán, L.: nEther: in-guest detection of out-of-the-guest

malware analyzers, *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, New York, NY, USA, ACM, pp. 3:1–3:6 (2011).

[17] King, S. T., Chen, P. M., Wang, Y.-M., Verbowski, C., Wang, H. J. and Lorch, J. R.: SubVirt: Implementing malware with virtual machines, *IEEE Symposium on Security and Privacy*, Vol. 0, pp. 314–327 (2006).

[18] Rutkowska, J.: Subverting VistaTM Kernel For Fun And Profit, *Black Hat Briefings* (2006).

[19] Ristenpart, T., Tromer, E., Shacham, H. and Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, New York, NY, USA, ACM, pp. 199–212 (2009).

[20] Chen, X., Andersen, J., Mao, Z., Bailey, M. and Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware, *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008.*, IEEE, pp. 177–186.

[21] Xiong, X., Tian, D. and Liu, P.: Practical protection of kernel integrity for commodity OS from untrusted extensions, *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, ISOC (2011).

[22] Payne, B., Carbone, M. and Lee, W.: Secure and flexible monitoring of virtual machines, *Annual Computer Security Applications Conference, 2007. ACSAC'07*, IEEE Computer Society, pp. 385–397 (2007).

[23] Sailer, R., Valdez, E., Jaeger, T., Perez, R., Van Doorn, L., Griffin, J., Berger, S., Doorn, L., Linwood, J. and Berger, G.: sHype: Secure Hypervisor Approach to Trusted Virtualized Systems, *IBM Research Report RC23511*, IBM Research Division (2005).

[24] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y. and Kato, K.: BitVisor: a thin hypervisor for enforcing i/o device security, *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, New York, NY, USA, ACM, pp. 121–130 (2009).

[25] Corporation, I.: Trusted Execution Technology (Intel TXT).
http://download.intel.com/technology/security/downloads/315168.pdf.

[26] Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R. and van Doorn, L.: vTPM: virtualizing the trusted platform module, *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, USENIX Association (2006).

[27] Sailer, R., Zhang, X., Jaeger, T. and van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture, *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, USENIX Association, pp. 16–16 (2004).

[28] Corporation, M.: BitLocker Drive Encryption.
http://windows.microsoft.com/en-US/windows7/products/features/bitlocker.

[29] Kauer, B.: OSLO: improving the security of trusted computing, *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association, pp. 16:1–16:9 (2007).

[30] Azab, A., Ning, P., Sezer, E. and Zhang, X.: HIMA: A hypervisor-based integrity measurement agent, *Annual Computer Security Applications Conference, 2009. ACSAC'09*, IEEE, pp. 461–470 (2009).

[31] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X. and Skalsky, N. C.: HyperSentry: enabling stealthy in-context measurement of hypervisor integrity, *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, New York, NY, USA, ACM, pp. 38–49 (2010).

[32] Litty, L., Lagar-Cavilla, H. A. and Lie, D.: Hypervisor support for identifying covertly executing binaries, *Proceedings of the 17th conference on Security symposium*, Berkeley, CA, USA, USENIX Association, pp. 243–258 (2008).

[33] Nipkow, T., Paulson, L. C. and Wenzel, M.: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS, Vol. 2283, Springer (2002).

[34] Bertot, Y. and Castéran, P.: *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*, Springer-Verlag (2004).

[35] Bevier, W. R.: Kit: A Study in Operating System Verification, *IEEE Transactions on Software Engineering*, Vol. 15, No. 11, pp. 1382–1396 (1989).

[36] Boyer, R. S. and Moore, J. S.: *A computational logic handbook*, Academic Press Professional, Inc., San Diego, CA, USA (1988).

[37] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: formal verification of an OS kernel, *Proc. of SOSP'09*, pp. 207–220 (2009).

[38] Jones, S. P.(ed.): *Haskell 98 Language and Libraries – The Revised Report*, Cambridge University Press, Cambridge, England (2003).

[39] Ball, T., Majumdar, R., Millstein, T. and Rajamani, S. K.: Automatic predicate abstraction of C programs, *Proc. of PLDI'01*, pp. 203–213 (2001).

[40] Henzinger, T. A., Jhala, R., Majumdar, R. and Sutre, G.: Lazy abstraction, *Proc. of POPL'02*, pp. 58–70 (2002).

[41] Clarke, Jr., E. M., Grumberg, O. and Peled, D. A.: *Model checking*, MIT Press, Cambridge, MA, USA (1999).

[42] Ball, T., Bounimova, E., Kumar, R. and Levin, V.: SLAM2: Static driver verification with under 4% false alarms, *Proc. of FMCAD'10*, pp. 35–42 (2010).

[43] Ball, T., Cook, B., Levin, V. and Rajamani, S. K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft, *Proc. of IFM'04*, pp. 1–20 (2004).

[44] Yang, J. and Hawblitzel, C.: Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System, *Proc. of PLDI'10* (2010).

[45] Barnett, M., Chang, B.-Y. E., DeLine, R., 0002, B. J. and Leino, K. R. M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs, *Proc. of FMCO'05*, pp. 364–387 (2005).

[46] De Moura, L. and Bjørner, N.: Z3: an efficient SMT solver, *Proc. of TACAS'08*, pp. 337–340 (2008).

[47] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S. and Chambers, C.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. of SOSP'95*, pp. 267–284 (1995).

[48] Nelson, G.(ed.): *System Programming in Modula-3*, Prentice Hall (1991).

[49] Hunt, G. C., Larus, J. R., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D. and Zill, T. W. B.: An Overview of the Singularity Project, Technical Report MSR-TR-2005-135, Microsoft Corporation (2005).

[50] Hejlsberg, A., Torgersen, M., Wiltamuth, S. and Golde, P.: *The C# Programming Language*, Addison-Wesley Professional, 3rd edition (2008).

[51] Morrisett, G., Walker, D., Crary, K. and Glew, N.: From System F to Typed Assembly Language, *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3, pp. 528–569 (1999).

[52] Maeda, T. and Yonezawa, A.: Writing an OS Kernel in a Strictly and Statically Typed Language, *Formal to Practical Security, LNCS 5458*, pp. 181–197 (2009).

[53] Maeda, T. and Yonezawa, A.: Writing practical memory management code with a strictly typed assembly language, *Proc. of SPACE'06* (2006).

[54] Maeda, T. and Yonezawa, A.: Typed assembly language for implementing OS kernels in SMP/multi-core environments with interrupts, *Proc. of SSV'10* (2010).

[55] Adve, S. V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *IEEE Comp.*, Vol. 29, No. 12, pp. 66–76 (1996).

[56] Boudol, G. and Petri, G.: Relaxed memory models: an operational approach, *Proc. of POPL'09*, pp. 392–403 (2009).

[57] Atig, M. F., Bouajjani, A., Burckhardt, S. and Musuvathi, M.: On the verification problem for weak memory models, *Proc. of POPL'10*, New York, NY, USA, ACM, pp. 7–18 (2010).

[58] International Organization for Standardization (ISO): Information Technology - Open Sys-

tems Interconnection - Security Frameworks in Open Systems - Part 3: Access Control, Technical report, ISO/IEC 10181-3 (1996).

[59] Bell, D. and LaPadula, L.: Secure computer systems: mathematical foundations and model, Technical Report M74-244, The MITRE Corporation, Bedford, MA (1973).

[60] Biba, K. J.: Integrity Considerations for Secure Computer Systems, Technical Report MTR-3153, The MITRE Corporation, Bedford, MA (1977).

[61] Clark, D. and Wilson, D.: A comparison of commercial and military computer security models, *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, IEEE Computer Society, pp. 184–194 (1987).

[62] Brewer, D. and Nash, M.: THE CHINESE WALL SECURITY POLICY, *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, IEEE Computer Society, p. 206 (1989).

[63] Ferraiolo, D., Cugini, J. and Kuhn, D.: Role-based access control (RBAC): Features and motivations, *Proceedings of 11th Annual Computer Security Application Conference*, pp. 241–48 (1995).

[64] Ferraiolo, D. and Kuhn, R.: Role-Based Access Control, *In 15th NIST-NCSC National Computer Security Conference* (1992).

[65] Sandhu, R., Coyne, E., Feinstein, H. and Youman, C.: Role-based access control models, *Computer*, Vol. 29, No. 2, pp. 38–47 (1996).

[66] Ferraiolo, D., Sandhu, R., Gavrila, S., Kuhn, D. and Chandramouli, R.: Proposed NIST standard for role-based access control, *ACM Transactions on Information and System Security (TISSEC)*, Vol. 4, No. 3, pp. 224–274 (2001).

[67] Sandhu, R. and Bhamidipati, V.: Role-based administration of user-role assignment: The URA97 model and its Oracle implementation, *Journal of Computer Security*, Vol. 7, No. 4, pp. 317–342 (1999).

[68] Sandhu, R., Bhamidipati, V. and Munawer, Q.: The ARBAC97 model for role-based administration of roles, *ACM Transactions on Information and System Security (TISSEC)*, Vol. 2, No. 1, pp. 105–135 (1999).

[69] Crampton, J. and Loizou, G.: Administrative scope: A foundation for role-based administrative models, *ACM Transactions on Information and System Security (TISSEC)*, Vol. 6, No. 2, pp. 201–231 (2003).

[70] Jha, S., Li, N., Tripunitara, M., Wang, Q. and Winsborough, W.: Towards formal verification of role-based access control policies, *IEEE Transactions on Dependable and Secure Computing*, pp. 242–255 (2007).

[71] Thomas, R. K. and Sandhu, R. S.: Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Autorization Management, *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI: Status and Prospects*, London, UK, UK, Chapman & Hall, Ltd., pp. 166–181 (1998).

[72] Becker, M., Fournet, C. and Gordon, A.: Design and Semantics of a Decentralized Authorization Language, *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, Washington, DC, USA, IEEE Computer Society, pp. 3–15 (2007).

[73] Halpern, J. and Weissman, V.: Using first-order logic to reason about policies, *ACM Transactions on Information and System Security (TISSEC)*, Vol. 11, No. 4, pp. 1–41 (2008).

[74] Weissman, V. and Lagoze, C.: Towards a Policy Language for Humans and Computers, *Research and Advanced Technology for Digital Libraries* (Heery, R. and Lyon, L., eds.), Lecture Notes in Computer Science, Vol. 3232, Springer Berlin / Heidelberg, pp. 513–525 (2004).

[75] Gofman, M., Luo, R., Solomon, A., Zhang, Y., Yang, P. and Stoller, S.: RBAC-PAT: A Policy Analysis Tool for Role Based Access Control, *Tools and Algorithms for the Construction and Analysis of Systems* (Kowalewski, S. and Philippou, A., eds.), Lecture Notes in Computer Science, Vol. 5505, Springer Berlin / Heidelberg, pp. 46–49 (2009).

[76] Guttman, J., Herzog, A., Ramsdell, J. and Skorupka, C.: Verifying information flow goals in security-enhanced Linux, *Journal of Computer Security*, Vol. 13, No. 1, pp. 115–134 (2005).

[77] Saltzer, J. and Schroeder, M.: The protection of information in computer systems, *Proceed-*

ings of the IEEE, Issue 9, Vol. 63, pp. 1278–1308 (1975).

[78] Lipton, R. J. and Snyder, L.: A Linear Time Algorithm for Deciding Subject Security, *J. ACM*, Vol. 24, pp. 455–464 (1977).

[79] Watson, R. N. M., Anderson, J., Laurie, B. and Kennaway, K.: Capsicum: practical capabilities for UNIX, *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, USENIX Association, pp. 3–3 (2010).

[80] Spencer, R., Corporation, S. C., Smalley, S., Loscocco, P., Agency, N. S. and Andersen, M. H. D.: The Flask Security Architecture: System Support for Diverse Security Policies, *Proceedings of the 8th USENIX Security Symposium*, pp. 123–139 (1999).

[81] Watson, R. N. M.: TrustedBSD: Adding Trusted Operating System Features to FreeBSD, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 15–28 (2001).

[82] KaiGai, K.: Security Enhanced PostgreSQL (2006).
http://code.google.com/p/sepgsql/.

[83] Macmillan, K., Brindle, J., Mayer, F., Caplan, D., Tang, J. and Technology, T.: Design and implementation of the SELinux policy management server, *Proceedings of the Security Enhanced Linux Symposium*, pp. 1–6 (2006).