Diagnosis Support of Embedded Systems based on Virtualization

Lei Sun[†] **and Tatsuo Nakajima**[††],

Department of Computer Science, Waseda University, Tokyo, Japan

Summary
In this paper, a runtime diagnosis infrastructure is presented for embedded systems. Different from existing methods of tracing system logs offline, our research focuses on analyzing system kernel data structures from runtime memory against predefined constraints periodically. The prototype system is developed based on a system virtualization layer, above on which the guest operating system and diagnosis services run simultaneously. The infrastructure requires few modifications to the source code of operating system kernel, thus it can be easily adopted into existing embedded systems for quick implementation. It is also fully software-based without introducing any specific hardware; therefore it is cost-efficient. The experiment results indicate that it can correctly detect several real world kernel-level security attacks with acceptable penalty to system performance.

Key words:
Security, diagnosis, embedded system, kernel data structures

1. Introduction

Recently ubiquitous computing is gaining significant interest in the field of embedded systems. More and more embedded devices have been used inside the current application-rich environment, with rapid development of various services, e.g. GPS, traffic navigation services etc. Mobile phone is a typical representative, also has become the most important information carrier. A trusted, robust and user-friendly mobile terminal is the fundamental of ubiquitous computing; therefore those embedded devices engaged in ubiquitous computing also require more security and robustness.

Modern embedded systems are also changing increasingly from specific-purpose to general-purpose. With their functionality on demand is growing, their software becomes more complicated as well. For instance, the lines of source code of current mobile phone is around 5-7 million, and keep growing. Numerous applications formerly developed for PC can be ported to embedded platforms more easily. Thus a plenty of open-source software have been introduced to mobile phones, e.g. Linux kernel and its upper layer applications. The opening of APIs and source code also bring security challenges to the design of embedded systems.

It is well known that there are many potential security risks in Internet. At the same time when embedded systems are designed more general-purpose together with open APIs, they are also more likely to suffer from security attacks. When embedded systems are under attack, it is often very difficult to diagnose. Moreover, in comparison with PC, embedded systems typically lack diagnosis utilities and administrative tools to pinpoint security problems. Meanwhile, ordinary users usually do not have enough technical knowledge to solve such problems. These problems may result in a negative influence of user experience with related products and manufactures. Therefore, there are great needs for runtime system-level diagnosis support for future advanced embedded systems.

However, existing research does not solve the problem very well yet. Hardware-based solutions [19] increase production costs of embedded systems, which are known sensitive to prices. Some solutions implemented as system kernel extensions or in application layers [1, 6, 11, 13-15] can be easily compromised by kernel level security attacks. Some offline solutions [13, 14, 20] seem effective, while they cannot cover runtime problems, hence they cannot fix them to help improve user experience.

To address above problems, we propose an innovative runtime diagnosis infrastructure for embedded systems. Its design is a balancing choice between security requirements of system design and practical engineering solutions. Current commodity operating systems like Windows, Mac OS and Linux are all written in C or C++; hence they cannot benefit from safe programming technique. Re-writing systems using safe programming languages, such as Java and C#, can remove almost 50% of existing related attacks [21]. But from engineering cost point of view, it is not practical at all; all legacy applications have to be rebuilt in the new system. Even if all these applications are open-source, developers also have to recompile and integrate them into a completely new execution environment. Therefore, we propose a diagnosis solution based on system virtualization technique to keep balance between system security and practice of innovative embedded system design.

The main contribution of this paper is twofold:

Manuscript received June 5, 2009
Manuscript revised June 20, 2009

- We propose a diagnosis infrastructure to protect the system from security attacks, whose attacking target is system kernel.
- We developed a prototype system based on a system virtualization layer to verify the feasibility of the proposed infrastructure. The experiments demonstrate that it can detect several known kernel-level security attacks with acceptable performance overhead.

The remainder of the paper is structured as follows: Section 2 describes related work, Section 3 contains the explanation of design issues and Section 4 presents system architecture and implementation. Section 5 is about evaluation, Section 6 discusses limitations and future work and Section 7 concludes the paper.

2. Related Work

In recent literature of system detection, some work has been done at different layers by using various methods. By connecting a specific hardware to the target system, Copilot [19] can detect kernel-level attacks by periodically checking kernel memory. Its monitoring task is deployed on an independent external PCI card. In its latest work, a constraint specification infrastructure [12] is proposed, which can be used to detect the inconsistency of kernel dynamic data. Gibraltar [7] compares values of kernel data structures in training and enforcement modes to detect kernel attacks using PCI network card.

With the virtualization technique has become popular, there are some virtual machine monitor (VMM) based solutions [10]. At kernel level, in certain signature-based intrusion detection systems [13], by hooking system calls, a large amount of log data is generated for the purpose of analysis, its volume of the offline log per day is about 1.2 GB [15]. Certain monitoring functions also have been implemented as kernel modules [11] which can be loaded into the monitored system. In the application layer, there are also some existing solutions to detect inconsistency of Linux kernel such as Chkrootkit [1], Rkthunter [6] and Tripwire [14]. They can check integrity of file systems or other critical system administrative binaries. But they can be easily cheated by kernel-level attacks by directly compromising the related kernel data, thus its monitoring cannot be trusted. Strider Ghostbuster [20] can detect all hidden files and processes for Windows by offline monitoring. It uses a cross diff-view based approach, which compares the view from user level with kernel level.

3. Design Issues

There are several general issues related with the design of diagnosis infrastructure. First is how to provide security isolation between the diagnosis service and the monitored target. Second is how to control the flexible granularity of isolation partition.

Security isolation can be provided by either hardware or software solutions. Recently virtual machine monitor (VMM) has become a popular alternative among software solutions. But virtual machine monitor-based systems are also facing security problems when their code size increases, which are mentioned in [16]. When virtual machine monitor suffers security attacks, the security isolation cannot be trusted; hence the services may also become unavailable. Our proposed diagnosis service is implemented based on microkernel architecture. These services are implemented as programs running on microkernel, separated from the guest operating system. The isolation is provided by several small code-size trusted computing bases (TCB), which can be trusted and believed bug-free. The microkernel can still restart them hence recover the whole system as its final solution, even when diagnosis service has crashed. Therefore, the microkernel-based system can still control the system when system services scale.

Furthermore, virtual machine only allows developers to divide the system at the level that includes both operating systems and their applications within each partition. Microkernel-based virtualization allows decomposing software systems at an arbitrarily fine level of granularity, allowing greater flexibility in optimizing that decomposition for the specific requirements of your application. The control of flexible granularity of partition is essential to fit various customized requirements for embedded products.

4. Implementation

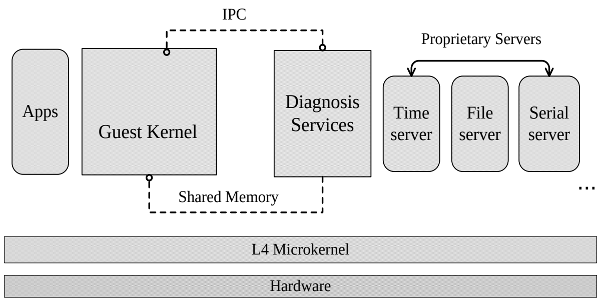

Fig. 1 System architecture.

In this section, we explain more details about system implementation. Our prototype system is developed based on system virtualization layer (also known as hypervisor) named L4 microkernel [17], whose purpose is to provide hardware abstraction and basic system services, including process scheduling and inter-process communication (IPC). Linux and diagnosis service are running on the system virtualization layer simultaneously. Fig. 1 shows the whole system architecture. The diagnosis service program can access Linux kernel memory to analyze kernel data structures at runtime.

4.1 Overview

The proposed infrastructure is based on the underlying system virtualization layer, which is in charge of scheduling among processes and inter-process communication (IPC). L4 microkernel virtualization layer provides strong isolation among the guest operating system kernel and upper layer service programs, which is highly reliable and believed bug-free. Linux applications provide main utilities to users, such as GUI, media file player, web browsing. Linux kernel is extended with diagnosis service programs above microkernel due to it are monolithic and lacks of security isolation mechanisms. The diagnosis services are to enhance the reliability of Linux kernel at runtime. Moreover, in the application domain, all Linux legacy applications can be reused in this infrastructure without any modification.

4.2 Diagnosis

In this section, we talk about how diagnosis is implemented. We choose Linux kernel data structures as our research objects, because of their close connection with system runtime states. We also explain how to select runtime monitored Linux kernel data structures among numerous candidates and how to write related constraint scripts used at system runtime.

4.2.1 Kernel data structures selection

Kernel data structures are used by operating systems to keep information about current system states. When changes happen within the system, related data structures are updated to reflect the current reality. For example, a *task struct* which represents a Linux task is created inside Linux kernel when a user launches a new application. The kernel scheduler also uses its related data fields to decide which the next process to dispatch is. Linux kernel data structures contain data, pointers and the addresses of kernel functions. Every Linux kernel data structure has close relation with a specific kernel sub-system such as process scheduling, memory management, file system etc.
Kernel data structures refer to certain specific data structures related with system resources not abstract data types in our research context. We focus on certain critical kernel data structures, which are related with runtime system resources, mainly CPU and memory resources. Moreover, there is a close relation between the granularity in the selection of these objects and the impact on the introduced system overhead. Currently we only choose mandatory kernel data structures related with system resources as our research objects as Table 1 shows.

Table 1: Selected kernel data structures

Name	**Category**	**Description**
task struct	CPU	processes
runqueue	CPU	process queue for scheduling
mm_struct	MEM	virtual memory of a process
vm_area_struct	MEM	an area of virtual memory
files struct	FILE	opened file descriptors
module	KMOD	loadable kernel module

We have also made a survey of kernel-level attacks to explore which data structures are more likely to be attacked. For example, in Table 1 the data structure of module is related with loadable kernel modules. Although kernel modules may not concern much with system resources, it is a very popular target to attack. Thus we also include it as a monitored object. Based on such supplement, the selected kernel data structures are expected to be system-critical and cover vulnerable security holes as many as possible.

4.2.2 Accessing kernel data structures

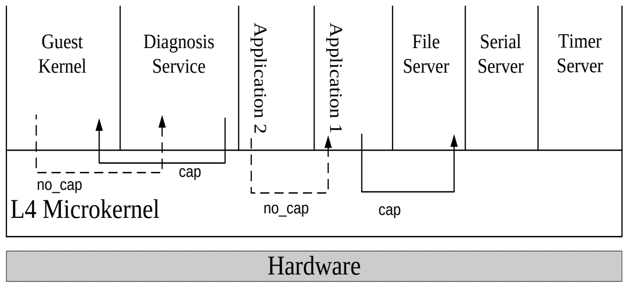

Fig. 2 A capability-based access control policy among memory sections

L4 microkernel virtualization layer manages a single address space shared among Linux kernel and other service programs, consist of several memory sections. Fig. 2 shows a capability-based privilege mechanism is introduced to manage the access control among individual memory sections. Linux kernel is executed as a normal program without any privilege. In our system, we assign the capability to the diagnosis service program, so that it can directly access the memory section of Linux kernel. We use the file *System.map*, generated when building Linux kernel to get runtime addresses of specific kernel

data structures. Thus we can directly access the address at runtime, dereference the pointer to get correct values of the corresponding data structure. For example, in the file *System.map* we can find the runtime address of *init_task* and *per_cpu_runqueues* respectively. By using them we can get system information about process scheduling and process management.
Type definitions of kernel data structures are used to describe their layouts inside runtime memory. We extract related type definitions by processing kernel source code using CIL (C Intermediate Language) [18] scripts automatically, which is a high-level representation along with a set of tools that permit analysis and source-to-source transformation of C programs. By using extracted type definitions, the diagnosis service can get correct runtime kernel information outside Linux kernel.

4.2.3 Diagnosis using dependency tree

The diagnosis service uses a dependency tree to maintain the relationships among system resources at runtime, such as parent-child relationships between processes, memory, owners. The dependency tree is updated by periodically reading the values of related kernel data fields explicitly. For example, we can know parent-child relationships between processes, by reading the data field of *children* in *task struct* without hooking fork system call. Comparing with the conventional method of hooking system calls, the kernel data structure-based method helps decrease the system overhead greatly. The dependency tree is used by the containment algorithm to identify possible malicious attacks. Table 2 shows the related kernel data structures and data fields used to maintain the system dependency tree.

Table 2: Correlation of kernel data structures

Dependency rule	**Data structures**	**Data field**
process/process	task_struct, runqueue	children
process/file	task struct, file struct	files, fs
process/user	task struct	uid, gid
process/memory	task struct, mm struct	mm, vma

Our research focuses on runtime diagnosis; hence the kernel data structures inside runtime memory are our main research objects. The dependency tree of kernel data structures contains rich runtime system information, including processes, their used memory, privilege and opened file descriptors. Hence diagnosis can be performed.

5. Evaluation

To evaluate the system, we set up experiments which are performed on a machine running the prototype system developed based on a L4 microkernel implementation L4Ka::Pistachio [3] and Iguana [2] from NICTA. It is a Dell Dimension 2400 machine, with 512MB RAM, equipped with a single 2.4GHz Pentium 4 processor running Linux kernel 2.6.13 as its guest operating system. Iguana [2] is designed for embedded systems, which supports various platforms, including ARM and IA32. For more convenience, we use the IA32 platform to evaluate it. Because the experiments focus mainly on overhead introduced by the diagnosis functions, there should not be much gap among different platforms.

5.1 Functional evaluation

We have implemented several runtime constraint scripts designed to protect kernel data structures of Linux kernel 2.6.13 using our system. We have tested them against implementations of existing Linux kernel attacks published in Packet Storm [4], which offers an abundant resource of security tools, exploits, and advisories. The tested security attacks are summarized in Table 3. There are mainly two methods of Linux kernel attacks, one is loadable kernel module (LKM) and the other is direct kernel memory exploit (KMEM).

Table 3: Linux kernel attacks survey

Name	**Attack method**	**Affected kernel data**
Knark-2.4.3	LKM	System call table
Adore-0.42	LKM	System call table
Modhide	KMEM	System call table
Adore-ng	KMEM	Kernel memory exploit
SuckIt-1.3	KMEM	Kernel memory exploit
Backdoor-caca	LKM	Interrupt descriptor table

According to the different attacking objects, they can be further summarized to two categories.

- *I*: Redirecting system control data, such as system calls, interrupt handlers or virtual file system functions to their own malicious routines, the representatives are Knark [8], Adore [5] and Backdoor-caca.
- *II*: Compromising system non-control data, such as direct kernel memory exploit, control of /proc file system, the example is SuckIt.

5.2 Analysis based on attacking model

False positives and false negatives are two common metrics to evaluate the accuracy of detection. False positives refer to those are legitimate operations that are marked tainted and reverted to a previous state, and false negatives refer to operations of attackers that are not caught.

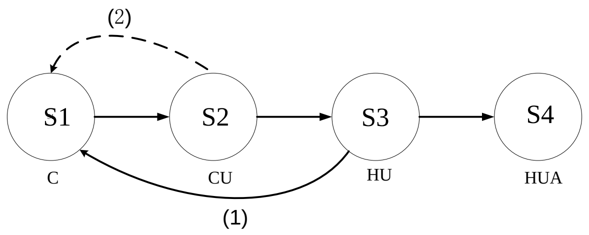

Fig. 3 A flow char for kernel attacks

Fig. 3 shows a flow chart for kernel attacks which consist of four states. *S1*: The system kernel is in a consistent state with several security holes. *S2*: Security holes have been found by attackers, are used to get root privilege. *S3*: Attackers have installed malicious kernel modules or compromised kernel memory directly to hide their own utilities, the system kernel has become inconsistent. *S4*: System is already comprised and used to attack other vulnerable hosts. And there are four symbols defined to illustrate states of the system kernel.

C: the system kernel is consistent

U: the system kernel is under attack

H: the system kernel is inconsistent due to hidden malicious utilities

A: the system is already in a active state to attack other vulnerable hosts

False positives. Fig. 3 shows that our system is designed to detect from kernel inconsistency for *S3* state marked as (1). While most of the previous application-level detection system focuses on *S2* state by hooking system calls or logging system events marked as (2). It is also the main difference between our research and previous research. In our system, if any kernel inconsistency has been detected, the kernel attacks can be concluded. Thus the false positive of our system is zero. In Fig. 3, suppose that the probability of each transition from *Si* to *Sj* is *pij* ($1 <= j < i <= 4$). The detection probability of kernel detection using our system is *p31*; the detection probability of application-level detection systems is *p21*. By using our kernel detection, the whole system detection accuracy will increase from *p21* to $p21 + (1 - p21) * p31$.

False negatives. Because our detection is based on periodical checking the update of kernel data structures, false negatives depend on the detection time interval. There is a trade-off between false negatives and system performance, especially to some transient attacks. If we shorten the detection time interval, the false negatives will decrease and result in the increase of system overhead. More details of performance analysis will be discussed in Section 5.3.

5.3 Performance analysis

In Section 5.1 we demonstrate the effectiveness of the diagnosis service, we measure the CPU overhead introduced by the diagnosis service in this section. We use runtime tracing mechanism provided by L4 microkernel to trace *switch_to* event hence evaluate the system overhead. The timestamps are filled with *rdtsc* instruction on IA32 platform.

Table 4: CPU consumption of main processes

Overhead (%)	**Name**	**Description**
66.668	L_timer	Linux interrupt handler
27.092	Kdbpoll	system kernel debugger
4.845	L syscall	Linux system call handler
0.559	monitor	runtime diagnosis services
0.33	serial	serial services
0.178	L1	application(bash)
0.0468	irq00	system irq handler
0.031	L17	application
0.0261	trace	trace buffer service
0.0237	timer	timer service
0.023	L18	application
0.0215	naming	naming service
0.0138	roottask	page fault handler

Table 4 shows the results of CPU consumption in system when diagnosis time interval is set to 800 milliseconds. It shows that the diagnosis service only consumes about 0.503% CPU resource, which is pretty lightweight. Linux kernel is implemented as two L4 processes: L timer acts as Linux interrupt irq handler, whose CPU consumption is about 66.67%; L_syscall is Linux system call handler, which consumes about 4.85%. The kdb poll task, which is in charge of kernel debug, consumes about 27%. We also change the time interval to check related CPU overhead introduced by the diagnosis service.

Table 5: Overhead changes along with interval and priority

Overhead (%)	**Timers (ms)**	**Priority**	**App priority**
0.5594384	800	100	99
1.1141184	400	100	99

2.2252043	200	100	99
4.4570269	100	100	99
8.8431218	50	100	99
0.5558166	800	200	99
0.5554242	800	110	99

Table 5 shows the overhead changes with detection intervals and the priority of diagnosis service. In the first five rows, the priority of diagnosis service keeps unchanged as 100, the priority of normal L4 application is 99. When the diagnosis interval changes from 800 milliseconds to 50 milliseconds, the introduced CPU overhead increases from 0.559% to 8.843%. It indicates the CPU consumption usage increases with the decrease of diagnosis interval time within the prototype system. It can be concluded that the maximum CPU overhead is 8.843% when the diagnosis interval is set to 50 milliseconds.
The first, sixth and seventh rows are group of comparative experiment. In these three rows, the diagnosis interval keeps unchanged as 800 milliseconds; the priority of diagnosis service is 100, 200 and 110 respectively. It indicates the CPU consumption usage varies slightly as 0.5594384%, 0.5558166% and 0.5554242%.

Table 6: Cycles used by detection

Category	**Detection** (cycle)	**Hz**
I	132	2.4 G
II	1409	2.4 G

To measure more fine-grained performance, we design several fault injection experiments, including overwrite system calls, overwrite IDT tables, direct modify task list to hide process etc. These fault injection experiments will trigger the invoking of the related diagnosis routines. As Table 6 shows, we observe two kinds of cases to perform fine-grained overhead analysis for detection based on kernel data structures. *I* stand for detection of system control data, *II* stand for detection of system non-control data. In *I*, only function pointer is compared with known good value; while in *II*, we have to traverse along multiple kernel data structures to verify its values against security specifications, such as relation between *run_queue* and *task_list*. The result shows that almost 10 times of CPU cycles are consumed in case *II* than case *I* in our platform.

6. Future Work and Discussion

Currently our prototype system only supports single processor, for further research it is planned to be extended to support multiprocessor architecture. The main challenge rises from the synchronization between kernel processes and diagnosis services. Our solution is to let diagnosis service evaluate based on the current snapshot of specific kernel data structures, at the same time kernel processes can still execute without any interference. Once inconsistency of kernel data structure has been detected, the modules may even recover them to certain consistent values. Some side effect may be introduced; currently related evaluation tests are still in progress.
Though our diagnosis service can detect several kernel-level security attacks, it also suffers from some limitations. We discuss about them as follows.
Our current research is based on the analysis of former known malicious attacks, suffers from arms race problems as well as other antivirus research work. Although the reverse engineering methods make our research effective to existing security case studies, they also limit our research to known problems, hard to find potential security holes. In our future work, we plan to combine static program analysis tools such as Daikon [9] to help us automatically explore potential system exploits inside Linux kernel.
Inside the proposed infrastructure, Linux kernel data structures are accessed from the diagnosis service, which is separated from kernel space. To correctly dereference pointers to kernel data structures, the diagnosis service has to know their layout inside memory from their definitions. Meanwhile, the definitions of related kernel data structures probably change during kernel development. Therefore the definitions of related kernel data structures should be updated according to their latest change, when we use the latest kernel source code.

7. Conclusion

As embedded systems are used increasingly for daily applications, ensuring automatic diagnosis and preventing the system from security attacks becomes even more important. In this paper, we have presented a runtime diagnosis infrastructure for embedded systems. A prototype system also has been developed to verify its feasibility based on security isolation provided by a system virtualization layer. The evaluation has demonstrated its effect of detecting the inconsistency of kernel data structures with low overhead. Moreover, its diagnosis does not require any modifications to the interfaces of former guest operating system calls, therefore they are expected to be easily applied to existing systems and reuse all of upper layer legacy applications.

Acknowledgments

This work was supported by CREST from Japan Science and Technology Agency (JST), through a grant of

Dependable Embedded Operating Systems for Practical Use.

References

[1] Chkrootkit. http://www.chkrootkit.org/.

[2]Iguana.http://www.ertos.nicta.com.au/software/kenge/iguana/project/latest/.

[3] L4Ka::Pistachio microkernel. http://l4ka.org/projects/pistachio/.

[4] Packet storm. http://packetstormsecurity.org/UNIX/penetration/rootkits/.

[5] The Adore rootkit. http://stealth.7350.org/rootkits/adore-ng-0.41.tgz.

[6] The Rootkit Hunter project. http://rkhunter.sourceforge.net/.

[7] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proc. of the 24th Computer Security Applications Conference (ACSAC)*, pages 77–86, Anaheim, CA, US, Dec 2008.

[8] J. R. Collins. Knark: Linux kernel subversion. Sans Institute.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec 2007.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. of the 10th Annual Network and Distributed Systems Security Symposium (NDSS)*, pages 191–206, San Diego, CA, USA, Feb 2003.

[11] R. K. Iyer, Z. Kalbarczyk, K. Pattabiraman, W. Healey, W.-M. W. Hwu, P. Klemperer, and R. Farivar. Toward application-aware security and reliability. *IEEE Security and Privacy*, 5(1):57–62, Jan 2007.

[12] N. L. P. Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. of the 15th USENIX Security Symposium*, pages 289–304, Vancouver, B.C., Canada, Aug 2006.

[13] R. A. Kemmerer and G. Vigna. Intrusion detection: A brief history and overview. *Computer*, 35(4):27–30, 2002.

[14] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proc. of the 2nd ACM Conference on Computer and Communications Security (CCS)*, pages 18–29, Fairfax, Virginia, US, Nov1994.

[15] S. T. King and P. M. Chen. Backtracking intrusions. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, Oct 2003.

[16] S. T. King, P. M. Chen, C. V. Yi-MinWang, H. J.Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 314–327, Oakland, CA, US, Mar 2006.

[17] J. Liedtke. On μ-kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, USA, Dec 1995.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Proc. of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, Mar 2002.

[19] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proc. of the 13th USENIX Security Symposium*, pages 179–194, San Diego, CA, US, Aug 2004.

[20] Y.-M.Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider Ghostbuster. In *Proc. of the 35th IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 368–377, Yokohama, Japan, Jun 2005.

[21] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *Proc. of the ACM workshop on Rapid Malcode (WORM)*, pages 11–18,Washington DC, US, Oct 2003.

Lei Sun received the B.E. and M.E. degrees in Computer Science from Tsinghua University, P.R. China in 2001 and 2004, respectively. Presently, he is a doctoral candidate of the Graduate School of Computer Science, Waseda University. His research interests include operating systems, embedded systems, and the security and reliability of systems. He is a student member of IEEE and ACM.

Tatsuo Nakajima is a professor in the Department of Computer Science, Waseda University. He was a researcher in School of Computer Science, Carnegie Mellon University in 1990-1993, a research engineer in AT&T Laboratories, Cambridge in 1998-1999 and a visiting research fellow in Nokia Research Center, Helsinki in 2005. He was also an associate professor in School of Information Science, Japan Advanced Institute of Science and Technology in 1993-1999. He was a program co-chair of RTCSA 2002 and ISORC 2003, and a general chair of RTCSA 2003 and ISORC 2005. His research interests include distributed systems, operating systems, ubiquitous computing, and information appliances.