Online Self-Diagnosis Self-Recovery Infrastructure for Embedded Systems

Lei Sun
System Platform Laboratories, NEC Corporation
l-sun@ap.jp.nec.com

Tatsuo Nakajima
Department of Computer Science, Waseda University
tatsuo@dcl.info.waseda.ac.jp

Abstract

Complexity of monolithic kernel of existing operating systems results in security exploits inevitably. When it is compromised, manual recovery from kernel-level attacks is usually time-consuming. The whole process is expected to be automatic and supported in system level. The requirement becomes more necessary for modern embedded systems, which lack such administrative and recovery tools for end users comparing with PC. We implement a prototype system called OSKROD to support system automatic recovery. OSKROD can take a collection of actions to recover infected kernel, after detecting kernel-level attacks infections based on system virtualization technique. Moreover, it can operate in two working modes: periodic mode and request-service mode, which can be fit in various application scenarios. Its recovery has been proved effective in fault injection experiments against real world attacks. The results indicate that it can correctly detect several kernel-level security attacks and recover the system with acceptable penalty to system performance.

Keywords: *security, recovery, detection, operating system.*

1. Introduction

Recently more and more embedded systems are engaged into the ubiquitous computing environment, which provides rich application services in our daily use. One of the most typical examples is the mobile phone. Currently people use it to make phone calls, listen to music, edit personal schedules, surf the web, do online shopping etc. It has become the most important information carrier. The mobile phone is also involved into a more intimate relationship with users as the increase of its popularity. Users intend to store sensitive and private data in the mobile phone rather than their personal computers, such as personal schedules, banking accounts even credit card information for online shopping. It is even expected to replace personal computers in the near future. Therefore existing embedded systems engaged in ubiquitous computing require more protection from security compromise.

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 source code of a modern mobile phone is around 5-7 million lines, and keeps growing. Numerous applications formerly developed for PC can be ported to embedded platforms easily with their open APIs. Thus a plenty of open-source software have been introduced to the mobile phone, e.g. Linux kernel and its upper layer applications. The opening of APIs and the source code also introduces great challenges to the design of secure embedded systems.

Nowadays Internet facing embedded systems are already under attack. For instance,

an almost 400% increase in mobile malware from 2005 to 2007 [1] is reported by F-Secure. Meanwhile, when embedded systems are designed more general-purpose together with open APIs, they are also more likely to suffer from security attacks. When such embedded systems are under attack, it is usually very difficult to detect and recover. Moreover, in comparison with PC, embedded systems typically lack diagnosis utilities and administrative tools to pinpoint security problems. Ordinary users do not have enough technical knowledge to solve above security problems. Therefore, there are great needs for automatic recovery requirements in system level for future advanced embedded systems.

Among numerous security attacks of embedded systems, kernel-level attacks (also known as kernel rootkits) are proved more difficult to detect and recover. These attacks directly compromise the operating system kernel, hide malicious utilities from administrative tools and hence launch back-door services for upcoming attacks. Latest kernel-level attacks for mobile phones already occur. Kernel rootkits change the behavior of kernels by modifying values of kernel data structures, forcing them to actively conceal the presence of adversary. Thus traditional application-level intrusion detection programs that depend on consistent kernel behavior cannot detect kernel rootkits.

It is difficult to detect anomaly behavior of operating system kernels still using traditional UNIX-like system architecture. It is obvious that you cannot trust detection results from the detected target itself. But the system virtualization technique can help on diagnosis and further recovery of the operating system kernel. System virtualization provides a mechanism to insert a software layer beneath an operating system kernel. Xen [2] and OKL4 [3] are two typical representatives of popular system virtualization solutions. The difference is that Xen is widely accepted for servers or clusters and OKL4 is for embedded systems. The system virtualization isolates security solutions from untrusted components, also enables them execute in the same or lower layer with the guest kernel. Comparing with former security solutions implemented in-kernel or as an application, the security solutions based on virtualization do not rely on kernel consistency. Therefore, security tools implemented in the virtualization layer are able to detect the execution of a vulnerable guest kernel.

We present OSKROD (Operating System Kernel Recovery on Demand), a virtualization-based prototype system, to provide automatic recovery to Linux kernel. OSKROD periodically examines the state of a running kernel against a collection of diagnostic specifications to detect the kinds of kernel rootkits. Upon detecting the corresponding infections, OSKROD invokes several actions to recover the system kernel.

The remainder of the paper is structured as follows: Section 2 describes related work, Section 3 analyzes the threat model and Section 4 presents system design and implementation. Section 5 is about case study, Section 6 talks about evaluation, Section 7 discusses about limitation and future work and Section 8 concludes the paper.

2. Related Work

Our related research can be mainly covered by detection and recovery two research domains. We give a brief introduction to each of them respectively.

In recent literature of system detection context, much research has been done at different layers by using various methods. By connecting a specific hardware to the target system, Copilot [4] can detect kernel-level attacks by periodically checking kernel memory. Its detection task is deployed on an independent external PCI card. In

its latest work, a constraint specification infrastructure [5] is proposed, which can be used to detect the inconsistency of kernel dynamic data. Gibraltar [6] 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 also rise some virtual machine monitor based solutions [7]. At kernel level, in certain signature-based intrusion detection systems [8], by hooking system calls, a large amount of log data is generated for the purpose of analysis, its volume of the off-line log per day is about 1.2 GB [9]. Certain detection functions also have been implemented as kernel modules [10] which can be loaded into the detected system. In the application layer, there are also some existing solutions to detect inconsistency of Linux kernel such as Chkrootkit [11], Rkthunter [12] and Tripwire [13]. 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 detection cannot be trusted. Strider Ghostbuster [14] can detect all hidden files and processes for Windows platform by off-line detection. It uses a cross view-diff based approach, which compares the view from user level with the one from kernel level. FACE [34] provides an integrated automated digital evidence discovery and correlation infrastructure. Based on XenAccess [35, 36], some secure related research is also deployed based on Xen virtual machine platform. In recently literature, SIM [37] benefits from hardware virtualization support to perform system monitoring while KOP [38] focuses more on kernel dynamic objects.

The research of recovery can be traced back to fault-tolerant and transaction-based systems. Roll-back recovery technique [15] uses snapshots at check points to perform recovery from some fatal errors. At each check point, the system makes a snapshot of some specific processes, which introduces the overhead to both CPU and memory resources. But this part of research is mainly used to enhance system availability, not designed for the purpose of system security. Some application-aware research [10] also has been deployed inside kernel, which is mainly used to address problems related with specific applications. Currently some safety-critical applications have adopted the data structure-repairing technical solutions [16] to help correctly represent the information that a program manipulates. Grizzard et al. [17] address the issue of recovering from attacks that hook the system call table by replacing the exploit with a clean copy. In [18] and [19], recovery solutions have been proposed, which are all offline recovery procedures, designed to recover the system call table or file systems.

3. Threat Analysis

The cause of security attacks of modern mobile phone can be mainly summarized as follows.

- First, difficulty of system upgrade may result in system security exploit. Because of either technical or non-technical reasons, to most of ordinary users, mobile phone is still hard to perform system upgrade frequently and conveniently. Linux kernel is known for its sophistication, mobile phone developed based on it may suffer from old kernel bugs and security holes.

- Second, openness introduces more challenges from the application domain. To shorten development period, more open source code have been introduced into mobile phone, e.g. Openmoko [20] uses Linux as operating system kernel, Apple has opened iPhone SDK [21] for application programmers. A lot of legacy applications from PC world can also run on mobile phone easily with little porting cost. Openness of system kernel and APIs makes

mobile phone more likely encounter security attacks.

- Third, powerful communication capability makes mobile phone always accessible for attackers. Mobile phone usually connects to Internet, cellar network and SMS service at the same time; it acts as a three-overlapping communication node. Users prefer their mobile phone always online, which greatly facilitate security attacks.

After extensive survey of security attack techniques, common kernel attack methods can be summarized as following steps: 1). Privilege lifting; 2). Object hiding; 3). System calls or interrupt handler hijacking; 4) Installing back door services.

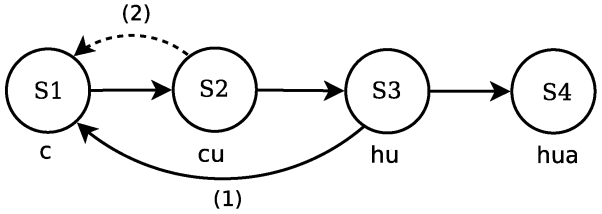

Figure 1. A flow chart of kernel attacks

We analyze the threat model for kernel attacks in more detail. Figure 1 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

Figure 1 shows that our system is designed to detect and recover 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), which is not suitable for embedded platforms.

4. System Design and Implementation

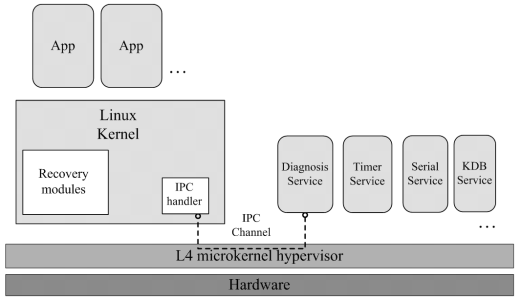

Figure 2. System architecture

In this section, we explain details about system design and implementation. Our prototype system is developed based on system virtualization layer (also known as system hypervisor) named L4 microkernel [22, 23], 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. Figure 2 indicates that the system consists of several recovery modules and a diagnosis service program. Recovery modules are implemented as several loadable kernel modules and the diagnosis service is implemented as a service program inside the microkernel system. The diagnosis service program can access Linux kernel memory to analyze kernel data structures at runtime, recovery modules are in charge of recovering kernel data structures inside kernel space. They can communicate via system IPC messages.

Figure 3 shows the components in our design. The system is composed of four main components: a specification manager, a detector, a recovery dispatcher and a recovery instance.

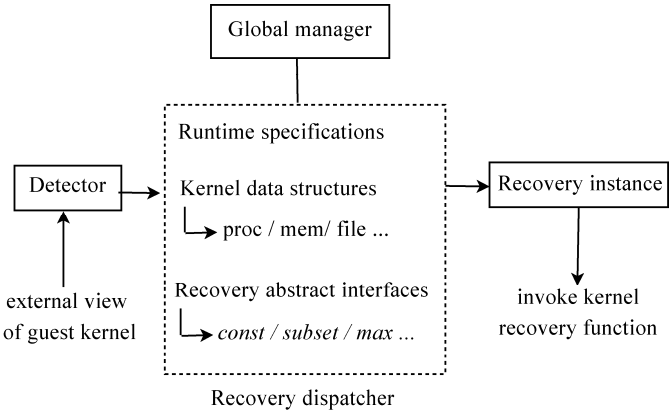

Figure 3. System components

1). Global manager is responsible for maintaining the global configurations; include runtime specifications, supported kernel data structures and recovery abstract interfaces. Specification is an assert-like expression on kernel data structures, e.g. the function pointer of a certain system call should be a constant value. Recovery abstract interfaces connect runtime specifications with the guest kernel dependent recovery instance. It is a wrapper layer.

2). Detector is a series of interfaces to be invoked at runtime to verify the system kernel against a given specification. Its implementation is based on underlying mechanism provided by system virtualization layer; hence from diagnosis program we can read Linux kernel memory directly.

3). Recovery dispatcher is managing the relations between runtime specifications and a set of recovery abstract interfaces when perform system recovery. In the example of the system call specification, the related recovery abstract interface is *const*; its recovery instance is to modify the function pointer of the system call to a known good value. Recovery dispatcher also supports multi-step recovery. If previous recovery action fails, it will invoke default recovery routines such as suspend or reboot the system.

4). Recovery instance is the guest system dependent recovery implementation. In our prototype, recovery instance for Linux is implemented as kernel modules inserted into the guest Linux kernel. The kernel has a IPC handler routine dealing with diagnosis services, which can be used to invoke recovery kernel functions and passing related parameters.

4.1. Policies of Virtualization Layer

Inside L4 microkernel, a fixed priority-based preempt scheduling algorithm [24] is implemented. The guest Linux kernel is developed as two processes, one is for handling interrupts and the other is used to handle system calls and maintain the update of kernel

data structures as well. In our prototype, the diagnosis service program are assigned higher priority than Linux kernel and Linux application processes, and the non-preempt flag is set to them. The non-preempt setting makes the runtime diagnosis not be interrupted by Linux kernel processes. At the same time, it also prevents guest kernel processes being interrupted in the middle of updating kernel data structures.

In the current implementation of L4 microkernel, the guest operating system kernel shares the same address space with other microkernel services. In our prototype, the diagnosis service program runs in the same address space as Linux kernel. However, there is a capability-based memory section management policy [25] supported in L4 microkernel. We assign the capability to diagnosis service but not to Linux kernel, so that the diagnosis service can access the memory address which belongs to Linux kernel freely, while Linux kernel can not access the memory address verse via.

4.2. External View of Guest Kernel

Filling the gap between external diagnosis services and guest operating system is one of the research challenges to virtualization based detection systems. The implementation of our prototype system relies on operating system level data structures to provide meaningful information. We use *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 *system.map* we can find runtime addresses of *init_task* and *per_cpu_runqueues* respectively. By using them we can get system information about process scheduling and process management.

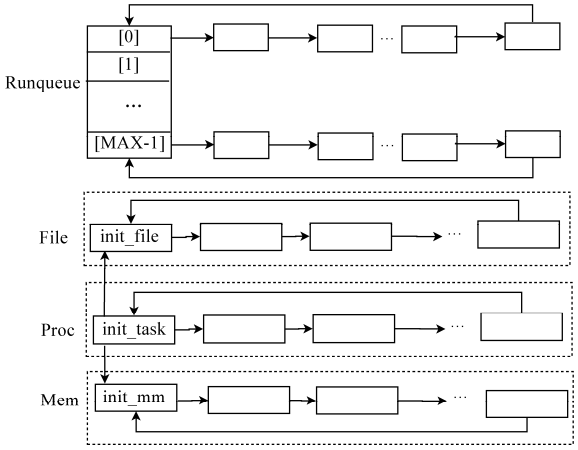

Figure 4. Runtime layout of process, memory, file descriptors and runqueue in kernel space

Type definitions of kernel data structures are used to describe their layout inside runtime memory. We extracted related type definitions by processing kernel source code using CIL (C Intermediate Language) [26] scripts automatically. By using extracted type definitions, the diagnosis service can get correct runtime kernel information outside Linux kernel. We can extract internal information of Linux kernel even outside it by casting above memory addresses. As Figure 4 shows, by accessing the exposed runtime memory address, diagnosis service can traverse the processes of Linux system outside Linux kernel.

4.3. Connecting Diagnosis to Recovery

The diagnosis service uses a dependency tree to maintain the relationships among system resources at runtime, such as parent-child relationships between processes, memory, and their privilege. The dependency tree is updated by periodically reading the values of related kernel data fields directly. 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 conventional methods of hooking system calls, it helps greatly decrease the system overhead. The dependency tree is used by the containment algorithm to identify possible malicious attacks. Table 1 shows the related kernel data structures and data fields used to maintain the system dependency tree.

Our research focuses on runtime diagnosis; hence the kernel data structures inside runtime memory are our main research objects. Figure 4 shows the runtime layout of selected kernel data structures, which are the data source of the kernel dependency tree. We can correlate among system resources based on its rich runtime system information. At runtime, we can periodically pick up a specification to verify the system or trigger a whole system detection and recovery in a request-service manner.

Table 1. Correlation of kernel data structure

Dependency rule	Data structure	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*

When the diagnosis service finds violation of a certain runtime specification of Linux kernel data structures, the recovery dispatcher will return a corresponding recovery abstract interface. Then underlying recovery instance will invoke the related recovery functions. In our prototype, recovery instance for Linux are implemented to perform fine-grained recovery of system kernel. These recovery kernel modules are organized according to their kernel functions, as process scheduling, memory management, kernel module or file descriptors.

The underlying recovery procedure can be summarized as follows. When a recovery function is invoked, the recovery dispatcher first sends parameters to Linux kernel process by IPC. When the IPC handler of Linux receives it, Linux kernel invokes a corresponding recovery function to modify internal state. After the recovery function is executed, Linux kernel returns the result to the recovery dispatcher using IPC. It verifies the return value, rechecks the state inside Linux kernel, and then decides whether to recover again if it fails.

5. Case Study

In this section, we will demonstrate the use and effectiveness of OSKROD for detecting and recovery kernel rootkits by presenting several case studies. Each of them is successfully detected and recovered by using abstract recovery interfaces of OSKROD.

5.1. Attack 1: Process Hiding

The process descriptors of all tasks running on Linux kernel 2.6 belong to a linked list called *all-tasks* list. This list contains process descriptors (*task_struct*) headed by the first process (*init_task*) created during system bootstrap. The all-tasks list is used by process

accounting utilities such as *ps*. While Linux scheduler uses another kernel data structure, called *runqueue*, to schedule processes for execution. The process hiding attack removes the process descriptor of a malicious process from all-tasks list, but it is still remained in *runqueue*. It results that the process is invisible to process accounting utilities, but it will still be scheduled for execution. In OSKROD, we can use subset recovery abstract interface to define *runqueue* is the subset of all-task list. If violation, the recovery function will be invoked, finally the hidden process will be removed from *runqueue*.

5.2. Attack 2: System Call Hijacking

System call hijacking is another widely used attack method by kernel rootkits. Although in Linux kernel 2.6, system call table is not exposed any more, attackers still can use loadable kernel modules or exploit */dev/kmem* to locate the address of a certain system call and hijack it to its own malicious routines. In OSKROD, we can use const recovery abstract interface to define known good values of addresses of system calls. If violation, the recovery function restores their original values.

5.3. Attack 3: Resource Exhausting

The applications of embedded systems are usually sensitive to the system resource. Meanwhile, the behavior of applications can also be defined by their resource consumption. OSKROD also supports recovery of resource exhausting. By using max recovery abstract interface, you can set the threshold of maximum memory consumption of a process. Therefore, we can detect resource exhausting attacks and hence recovery the system by restarting the related processes.

6. 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 [27] and Iguana [28, 29] 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 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 and recovery functions, there should not be much gap among different platforms.

6.1. Functional Evaluation

Table 2. 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 description table

We have implemented several runtime specification scripts designed to perform detection and recovery for Linux kernel 2.6 using our system. We have tested them against several existing Linux kernel attacks published in Packet Storm [30], which offers an abundant resource of security tools, exploits, and advisories. The tested security attacks are summarized in Table 2. There are mainly two methods of Linux kernel attacks, one is loadable kernel module (LKM) and the other is direct kernel memory exploit (KMEM). They can be further summarized to two categories.

- I: Redirecting system control data, such as system calls, irq handlers or virtual file system functions to their own malicious routines, the representatives are Knark [31], Adore [32] and Backdoor-caca.
- II: Compromising system non-control data, such as direct kernel memory exploit, control of /proc file system, the example is SuckIt.

6.2. Performance Analysis

Table 3. CPU consumption of main processes

Percentage (%)	Name	Description
65.668	*L_timer*	Linux interrupt handler
26.092	*kdbpoll*	System kernel debugger
4.845	*L_syscall*	Linux system call handler
2.259	*diagnosis*	Runtime diagnosis service
0.330	*serial*	Serial service
0.178	*L1*	Application(bash)
0.047	*irq00*	System irq handler
0.031	*L17*	Application
0.026	*trace*	Trace buffer service
0.024	*timer*	Timer service
0.023	*L18*	Application
0.022	*naming*	Naming service
0.014	*roottask*	Page fault handler

*idle process is counted but not showed.

In Section 6.1 we demonstrate the effectiveness of the recovery 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 3 shows the detail of CPU consumption in system when diagnosis time interval is set to 250 milliseconds. It shows that the diagnosis service only consumes about 2.259% CPU resource, which is pretty lightweight. Linux kernel is implemented as two L4 processes: L timer acts as Linux *irq* handler, whose CPU consumption is about 65.67%; *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 26%.

6.2.1. Diagnosis Performance: We also change the time interval to check related CPU overhead introduced by the diagnosis service. To give a fair statistic result, we define the measurement as long as we can, the sample data is dumped every 1,000 times context switch. Table 4 shows the overhead changes with detection intervals. The priority of diagnosis service and Linux *irq* handler process keeps unchanged as 100, the

priority of Linux system call hander process 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%

Table 4. Overhead changes along with interval

Overhead (%)	Time interval (ms)
0.559	800
1.114	400
2.225	200
4.457	100
8.843	50

*priority of diagnosis server and Linux irq handler are 100, priority of Linux syscall handler is 99.

To observe the relationship between introduced overhead and the priority of diagnosis service, we design some other experiments. During the experiments, the priority of Linux application process is 98, Linux system call handler process is 99 and Linux interrupt handler is 100. We change the priority of diagnosis service from 100 to 110 and 200; repeat three kinds of experiments 10 times. Stress workload experiment is also measured when the priority of diagnosis service is set to 100 by using *stress –cpu 8 –timeout 1000s*. From Table 5, it concludes that the overhead varies little when the process priority changes and it is also insensitive to the workload.

Table 5. Overhead changes along with priority and workload

Overhead (%)	Priority of diagnosis service
0.555	200
0.556	110
0.559	100

*priority of Linux irq handler is 100, priority of Linux syscall handler is 99.

We also design experiments to get CPU cycles used for diagnosis. It is about 1.605 million CPU cycles, which means the whole diagnosis will be finished within 1 second for a 1.6 MHz platform. To those platforms with slow processors, the diagnosis and recovery is supported in a request-service mode. When the embedded products are charged with AC adaptor, the diagnosis and recovery service can be enabled and provide recovery on demand.

6.2.2. Recovery Performance: To measure the performance of recovery, we design several fault injection experiments, including hooking system calls, overwriting irq tables, direct modifying *all-task* list to hide processes etc. These fault injection experiments result in invoking the recovery functions in kernel space. As Table 6 shows, we observe two kinds of cases to perform fine-grained overhead analysis for detection and recovery based on kernel data structures. I stands for system call hooking case studies, II stands for hidden process detection and recovery cases. For the detection side, in I, only function pointer is compared with known good value; while in II, we have to traverse along runtime kernel data structures to verify its values against runtime specifications, such as the subset relation between *runqueue* and *all-task* list. It results that almost 10 times of CPU cycles are used in case II

than case I. For the recovery side, both cases shows recovery consumes much more CPU cycles comparing with detection. This is due to mutual protections and lock mechanisms of Linux kernel, which also result in blocking inside the microkernel environment. While the CPU cycles consumed by recovery depend on corresponding recovery functions, general conclusions can not be drawn from the experiments.

Table 6. CPU cycles used for recovery and its percentage

Category	Detection	Recovery	R/(R+D) (%)
I	132	3846648	99.997
II	1409	50816	97.302

7. 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 security research. Although 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 [33] to help us automatically explore potential system exploits inside Linux kernel.

Inside our prototype system, Linux kernel data structures are accessed from the diagnosis service, which is separated from kernel space. To dereference pointers to kernel data structures correctly, 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.

8. Conclusion

As embedded systems are used increasingly for daily applications, ensuring automatic diagnosis and recovery to protect system from security attacks becomes even more important. In this paper, we have presented an automatic diagnosis recovery 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 guaranteeing the consistency of kernel data structures and performing fine-grained system recovery. Moreover, its diagnosis and recovery service can support multiple working modes. Therefore, it is expected to be easily applied to various scenarios for embedded systems.

Acknowledgment

This work was supported by CREST project from Japan Science and Technology Agency (JST), through a grant of Dependable Embedded Operating Systems for Practical Use.

References

[1] "M. Hypponen. The state of cell phone malware in 2007." http://www.usenix.org/events/sec07/tech/hypponen.pdf.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A.Warfield, "Xen and the art of virtualization," Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, USA, pp.164–177, Oct 2003.

[3] "Open Kernel Labs: OKL4." http://www.ok-labs.com/products/okl4.

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

[5] 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," Proc. of the 15th USENIX Security Symposium, Vancouver, B.C., Canada, pp.289–304, Aug 2006.

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

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

[8] R.A. Kemmerer and G. Vigna, "Intrusion detection: A brief history and overview," Computer, vol.35, no.4, pp.27–30, 2002.

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

[10] 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, vol.5, no.1, pp.57–62, Jan 2007.

[11] "Chkrootkit." http://www.chkrootkit.org/.

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

[13] G.H. Kim and E.H. Spafford, "The design and implementation of tripwire: a file system integrity checker," Proc. of the 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, US, pp.18–29, Nov 1994.

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

[15] M. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson., "A survey of rollback-recovery protocols in message-passing systems," ACM Computing Surveys, vol.34, no.3, pp.375–408, Sep 2002.

[16] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA), Anaheim, CA, US, pp.78–95, Oct 2003.

[17] J.B. Grizzard, J.G. Levine, and H.L. Owen, "Reestablishing trust in compromised systems: recovering from rootkits that trojan the system call table," Proc. of the 9th European Symposium on Research Computer Security (ESORICS), French Riviera, France, pp.369–384, Sep 2004.

[18] N. Zhu and T.C. Chiueh, "Design, implementation, and evaluation of repairable file service," Proc. of the 33th IEEE International Conference on Dependable Systems and Networks (DSN), San Francisco, CA, US, pp.217–226, Jun 2003.

[19] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara., "The taser intrusion recovery system," Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP), Brighton, UK, pp.163–176, Oct 2005.

[20] "Openmoko." http://wiki.openmoko.org/wiki/MainPage.

[21] "iPhone SDK for iPhone OS 3.0 beta." http://developer.apple.com/iphone/program/sdk.html.

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

[23] J. Liedtke, "Improving IPC by kernel design," Proc. of the 14th ACM Symposium on Operating System Principles (SOSP), Asheville, NC, USA, pp.175–187, Dec 1993.

[24] "NICTA L4-embedded Kernel Reference Manual." http://www.l4ka.org.

[25] "Iguana User Manual," 2005. http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/userman.pdf.

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

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

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

[29] G. Heiser, "The role of virtualization in embedded systems," In Proc. of the 1st workshop on Isolation and Integration in Embedded Systems, Glasgow, Scotland, pp.11–16, Apr 2008.

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

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

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

[33] 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, vol.69, no.1–3, pp.35–45, Dec 2007.

[34] A. Case, A. Cristina, L. Marziale, G. G. Richard III, and V. Roussev, "FACE: Automated Digital Evidence Discovery and Correlation," In Proceedings of the 8th Annual Digital Forensics Research Workshop (DFRWS 2008), Baltimore, MD, 2008.

[35] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," In Proceedings of The 2008 IEEE Symposium on Security and Privacy, Oakland, CA, May 2008.

[36] Bryan D. Payne, Martim Carbone, and Wenke Lee, "Secure and Flexible Monitoring of Virtual Machines," In Proceedings of The 23rd Annual Computer Security Applications Conference (ACSAC 2007), Miami Beach, FL, December 2007.

[37] Monirul Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," In Proceedings of The 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, IL, November, 2009.

[38] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang, "Mapping Kernel Objects to Enable Systematic Integrity Checking," In Proceedings of The 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, IL, November, 2009.

Authors

Lei Sun received the B.E. and M.E. degrees in Computer Science from Tsinghua University, P.R. China in 2001 and 2004, respectively. He received the Ph.D. degree in Computer Science from Waseda University, Japan in 2010. Presently, he is a researcher in System Platform Laboratories, NEC Corporation. His research interests include distributed systems, operating systems, embedded systems, and the security and reliability of systems. He is a 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. His research interests include distributed systems, operating systems, ubiquitous computing, and information appliances.