Journal of Information Processing Vol.21 No.4 650–659 (Oct. 2013)

[DOI: 10.2197/ipsjjip.21.650]

Regular Paper

An Extensible Secure OS Architecture for Embedded Systems

Ning Li[1,a)] Yuki Kinebuchi[1,b)] Hiromasa Shimada[1,c)] Tatsuo Nakajima[1,d)]

Received: November 30, 2012, Accepted: June 14, 2013

Abstract: Some recent researches have shown that using a monitoring service outside the target system above hypervisors is an efficient way to protect the target system. The hypervisors isolate the monitoring service based on MMU-methods to improve security. However, The MMU-method may cause heavy overhead when there is no hardware support, which makes this method not viable for embedded processors that are rarely equipped with hardware virtualization extensions. In addition, the vulnerabilities that exist in hypervisors may compromise the isolation. In this paper, we propose a secure OS architecture that fits embedded systems without the dependency of a hypervisor. It provides a robust isolation between the monitoring service and the guest OS based on local memory, a hardware feature. In order to generalize this architecture, we adopt a secure pager to extend the local memory space (physically small) virtually by a swap mechanism with integrity checking of the monitoring service. The secure pager can also update the monitoring service to extend monitoring functions without disturbing the running of the guest OS. Comprehensive evaluations are made in our framework with one instance of embedded Linux as the guest OS and an isolated monitoring service running with the secure pager. The results demonstrate functions of the secure pager and influence of the secure pager on Linux in our system. On processors with a proper architecture, we can build an extensible secure OS architecture with reasonable resource consumption, without the issue of heavy overhead to the guest OS.

Keywords: secure architecture, embedded systems, multi-core

1. Introduction

The security of operating systems (OS) is important in embedded, desktop and server environments in recent years. The traditional secure solution is to add malware detection tools into the system. However, this method might be unable to detect the malware that can crack the kernel and hide itself from the detection tools, and the detection tools may also be compromised by attacks from the malware [5]. Therefore, system researchers have started to investigate other methods to solve this problem.

An efficient solution in existing researches shown in **Fig. 1** is to move the malware detection tools outside the system to avoid attacks from the infected system. We call these tools monitoring services in this paper for the reason that they run as services that have monitoring functions. However, the isolation between the monitoring service and the vulnerable system is important to security.

Many approaches solve this problem based on hypervisors. These approaches can be divided into two groups according to the type of monitoring services: passive monitoring and active monitoring. References [7], [11], [12] use passive monitoring services in their systems. Passive monitoring cannot protect the target OS against attacks. It only checks if the target OS has already been infected. This causes little overhead to the target OS. On the other hand, there are also some other researches [8], [20] using active monitoring. They place hooks into the target OS to obtain necessary information for protecting it from attacks. However, this active style leads to visible overhead from system route changes between the target OS and the monitoring service. The hypervisor can isolate the monitoring service with the MMU-method [2], [19], [23]. References [25], [26] shows that in x86 processors, MMU virtualization in hypervisors without hardware support would cause heavy overhead to performance. Since hardware support of MMU virtualization is rarely provided in embedded processors, we consider that MMU virtualization by software would cause visible overhead in embedded systems.

There are also some approaches providing hardware-assisted isolation without the dependency of hypervisors, such as *HyperSentry* [3], *LMEM* method [13] [*1] and TrustZone [1] equipped in ARM processors. Along with more functions and higher complexity have been imported into hypervisors, some vulnerabilities have also been brought into them [9]. CVE (Common Vulnerabilities and Exposures) reports that in Xen, there are 39 vulnerabilities in 2012 and 4 vulnerabilities in 2011 [6], and several researches pay attention to improve the security of hypervisors, such as a prototype using Xen [16] and *HyperSafe* [29]. While another research does not focus on enhancing the reliability of the hypervisor, and uses hardware-assisted methods to protect guest OSes from the possible infected hypervisor [24]. Several steps

1 Department of Computer Science and Engineering, Waseda University, Shinjyuku, Tokyo 169–8555, Japan
a) lining@dcl.cs.waseda.ac.jp
b) yukikine@dcl.cs.waseda.ac.jp
c) h-shimada@dcl.cs.waseda.ac.jp
d) tatsuo@dcl.cs.waseda.ac.jp

*1 Basis of big local memory space that is unsuitable for embedded systems.

© 2013 Information Processing Society of Japan

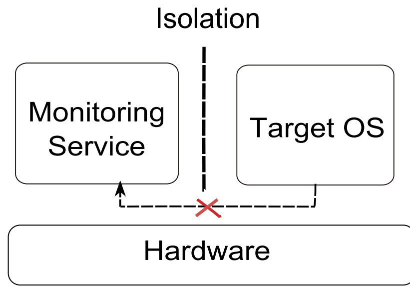


Fig. 1 Outside monitoring.

are still needed to be done to build a secure hypervisor environment. Therefore, the isolation based on the hypervisor might be compromised, and providing isolation without the dependency of hypervisors can solve this problem efficiently to protect the monitoring service.

In this paper, we propose a secure architecture for embedded systems that isolates the monitoring service based on a hardware feature, local memory, without the dependency of hypervisors. The secure pager implemented in our system can extend the size of the local memory virtually and update the monitoring service to extend monitoring functions without blocking the guest OS. In our system, the monitoring service can be executed in local memory to avoid attacks from the compromised guest OS. This extensible architecture can improve reliability with reasonable resource consumption on processors with a proper architecture.

The rest of the paper is structured as follows. In Section 2, the whole design method, an overview of the system architecture and security of the proposed framework are presented. In Section 3, we explain necessary details of implementation of system functions and schemes. Section 4 contains evaluations of the system and analysis about the architecture. In Section 5, our paper is concluded and in Section 6, future directions are discussed.

2. Design Issue

In this section, we will firstly introduce the basic principle of the isolation based on local memory. Then, an overview of our system with the isolation is introduced. Finally, we analyze security of the proposed framework and give a comparison with another similar research.

2.1 Isolation Based on Local Memory

Local memory is a programmable memory region for its own core, and cannot be accessed by other cores in the same processor. If one multi-core processor is equipped with this feature, it can be applied to provide isolation by running the monitoring service in one core's local memory and executing the guest OS upon other cores. The guest OS running on other cores cannot touch this core's local memory (content of the monitoring service). We also assume that this core is isolated by hardware configuration from other cores, so that they cannot get control of the core running the monitoring service to make attacks. More details about the machine architecture with local memory are presented in Ref. [14].

There have already been some processors with a similar architecture that can be seen as local memory, such as the Cell Broadband Engine (Cell BE) [22] and the SH-4A processor in RP1 [31].

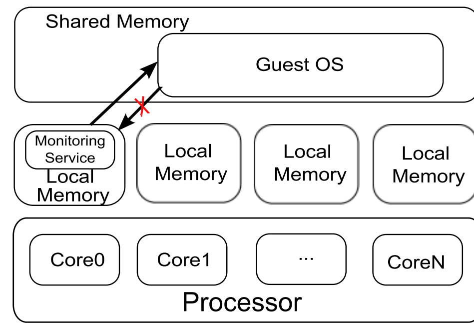


Fig. 2 Prototype system overview.

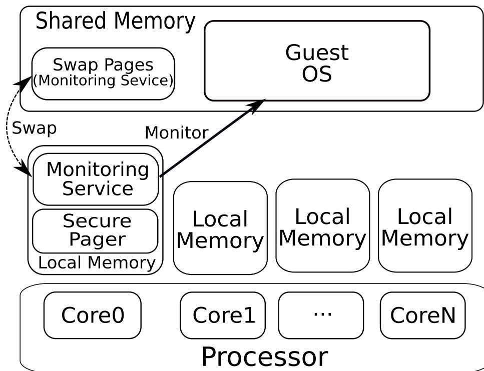


Fig. 3 System overview.

If an on-chip memory region can be restricted to be accessed by only one core, this region can be used as local memory. Intel has announced a type of processor called Single-Chip Cloud Computer (SCC) [4] that deals with the cache with software-managed coherency. The private cache for respective core should be able to be treated as local memory. We can foresee that along with an increasing number of cores in future processors, software-managed coherency is preferred than hardware cache coherency, because the latter one needs more lines to connect cores together, which would take more processor dies. With the private cache equipped for software-managed coherency, these processors can provide isolation with the local-memory method.

2.2 System Overview

With the local memory, the prototype overview of our system is shown in **Fig. 2**. In our system, the monitoring service runs in one core's local memory, and the guest OS runs on other cores. The monitoring service can monitor the state of the guest OS, and the guest OS are not permitted to read or modify the content of the monitoring service.

2.2.1 Local Memory Space Extension

However, we cannot expect that the size of the local memory (on-chip memory or private cache) would be big, and the monitoring service may need more memory space to be executed. In order to generalize this architecture, we adopt a secure pager to extend the local memory space virtually with a swap mechanism between shared memory and the local memory in **Fig. 3**. The pages can be swapped in and out to execute the monitoring if needed.

However, both the secure pager and the guest OS can access shared memory. If the content of the monitoring service stored in

© 2013 Information Processing Society of Japan

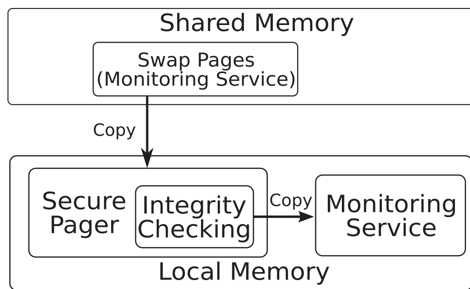


Fig. 4 Secure pager.

shared memory is modified by the guest OS, the monitoring service may be unable to correctly complete monitoring functions. Therefore, when the monitoring service needs to be loaded into the local memory, the secure pager that exists in the local memory will check the integrity of the content in **Fig. 4**. If the integrity is not corrupted, the content is copied and the monitoring service continues to run. Otherwise, it shows that our system has been infected. The secure pager will reboot the whole system to ensure the reliability of the monitoring service.

2.2.2 Extensibility

When we need to fix bugs, make optimizations or add new features to the monitoring service, the normal running of the guest OS is not expected to be disturbed. Automatic update is a good method to solve this problem. The main point of the automatic update is the security. The update procedure needs to be safe enough to make sure that the monitoring service is updated correctly. We add an automatic update function to the secure pager, which is trustworthy for permanently existing in the local memory. Details of this method will be discussed in the implementation part. With this function, we can extend functions of the monitoring service freely for making optimizations or adding features.

2.3 Security of Framework

In our framework, the vulnerable guest OS running user applications may be compromised by malware. The monitoring service checks the guest OS whether it is running in an abnormal state. The monitoring service stored in shared memory may be read or modified by the vulnerable guest OS. The secure pager based on the local memory isolates the monitoring service at runtime and verifies the integrity of the monitoring service that is swapped into the local memory to ensure the monitoring functions.

Many monitoring technologies can be used in our approach. Because we focus on the architecture to isolate the monitoring service, we will not discuss the monitoring technology in details. We use a sample monitoring service in our prototype framework. If the secure pager detects corruption of the integrity of the monitoring service, it shows that the guest OS is compromised. This architecture cannot recover the monitoring service efficiently unless the monitoring service running in the local memory could restore the guest OS to the normal state. In this case, the secure pager will reboot the whole system to restore to the normal state.

The integrity checking can ensure that the monitoring service runs correctly in the local memory, which can effectively improve the reliability of the monitoring service. However, the secure pager only detects the modification on the monitoring service stored in shared memory. The content may be read by the compromised guest OS to reveal messages. In order to avoid the "read attacks," the encryption method can be added to the content of the monitoring service stored in shared memory. The monitoring service would be encrypted in shared memory and decrypted when it is loaded into the local memory. Obviously, the encryption method would cause overhead during the running of the monitoring service. More details will be discussed in the implementation part.

There is another research *HyperSentry* [3] that uses hardware-assisted methods and provides a software component that is properly isolated from the hypervisor to enable stealthy and in-context measurement of the runtime integrity of the hypervisor. It uses some similar technologies with our approach, such as hardware-based isolation and integrity checking with hash algorithms. We will give a comparison between *HyperSentry* and our approach. Firstly, *HyperSentry* basically faces to server processors, while we focus on applying our approach to embedded systems. Secondly, *HyperSentry* blocks the whole system during the integrity checking and uses a remote verifier to check the integrity evidences. In our approach, we use a special core for monitoring tasks, and can do the security checking and update the monitoring service without disturbing the normal running of the guest OS. Finally, *HyperSentry* does the integrity checking with isolation provided by the cooperation of an out-of-band channel and System Management Mode equipped on the processor. Our approach focuses on the reliability of the monitoring service with isolation basis of the local memory, a simple hardware feature.

3. Implementation

In this section, firstly we introduce the implementation environment of our system. Then, integrity checking, execution and automatic update of the monitoring service are illustrated. Finally, the system architecture with two schemes is proposed.

3.1 Implementation Environment

We choose a development board RP1 [31] that owns a SH-4A processor with 4 cores equipped with respective 128 Kbytes user memory, which can be treated as local memory.

RP1 is a development board produced by Renesas Electricity Cooperation for use in multimedia equipment, network and other applications in embedded systems. It owns 4 cores running at 600 MHz, and each core incorporates an FPU, a CPU and an MMU, which is equipped with a 4-entry fully associative instruction TLB and a 64-entry fully associative unified TLB.

Because embedded Linux has already supported SH-4A architecture, we use one instance of embedded Linux with a kernel version of 2.6.16, which uses a network file system exported by a remote host machine, as the guest OS.

Since we focus on applying our architecture to embedded systems, maybe real-time systems, it is better to use a monitoring service with the passive pattern to reduce the overhead of the guest OS. We choose a passive monitoring service that can check the entries of the Linux system call table and detect the *hide_task*

© 2013 Information Processing Society of Japan

rootkit that uses the DKOM (Direct Kernel Object Manipulation) mechanism to infect the kernel data [21].

The host machine is running Ubuntu 12.10 with a quad-core Intel Core 2 processor Q9400 of 2.66 GHz, 4 GB RAM and 320 GB hard disk. We use it for compiling the system boot image and completing the encryption part in the automatic update.

3.1.1 Boot Mechanism

RP1 can directly load the system boot image file into an assigned address via network from the remote host machine. When RP1 boots, firstly, the boot image is loaded into a fixed address. Then the boot loader in the system image will relocate the secure pager into the local memory and relocate xv6 and Linux into shared memory. Linux does not boot until the secure pager starts to execute the monitoring service. Finally, Linux starts to boot to run user applications.

During this procedure, firstly we assume that the host machine, which can be managed well to ensure the security, and the RP1 firmware are trustworthy. The communication between the host machine and RP1 can be protected by existing technologies (not implemented in current work), such as SSL [27]. It is considered that the boot image can be loaded into shared memory correctly. We also assume that the only attacker is the vulnerable Linux. Because Linux does not start to boot until the loading and the booting of the secure pager are completed, the loading process of the secure pager is unable to be compromised by Linux. The system can boot to the normal working state.

In the following sections, we will introduce functions of our system.

3.2 Integrity Checking of Monitoring Service

In this section, we propose how the integrity of the monitoring service is checked. As we discussed in Section 2.2.1, when the monitoring service is loaded into the local memory, the integrity needs to be verified whether the content is modified or not.

Hash algorithms are popular for integrity checking of digital content. They are one-way mathematical algorithms that take an arbitrary length input and produce a fixed length output string. A hash value of a fixed data segment is unique, and it is almost impossible to find two different data strings with the same value. Besides, the hash values themselves are small and take little space in the local memory. It is suitable for integrity checking in our system.

We apply MD5 and SHA1 hash algorithms in the system. MD5 is not so robust in Ref. [28]. However, we use MD5 here to compare with SHA1 that has a more complex algorithm and better security.

The mechanism of the checking procedure shown in **Fig. 5** is introduced as following:

- Firstly, hash values of all the pages of the monitoring service are calculated by the secure pager and stored into the hash table that exists permanently in the local memory. Then, the monitoring service is encrypted, and the secure pager starts to execute the monitoring service.
- When a page of the monitoring service is needed for running, it will firstly be copied into the local memory and decrypted. The hash value of this page is calculated again by the secure pager, and compared to the value stored in the hash table. If they are different, it shows that the monitoring service is compromised, and the whole system needs to reboot to ensure that the system is running in a trustworthy state. Otherwise, the secure pager will set the TLB mapping for executing the monitoring service.
- When all the space in the local memory is used and there needs to load in a page for the monitoring service, the secure pager will choose a page in the local memory and swap it out. Because this page's content may change during the running of the monitoring service after it is loaded into the local memory and in this case, it is different from the original page stored in shared memory. The secure pager will recalculate the hash value, update the value stored in the hash table, encrypt this page, copy it back into shared memory and free this page. Then, the needed page can be loaded into the local memory as the previous step.

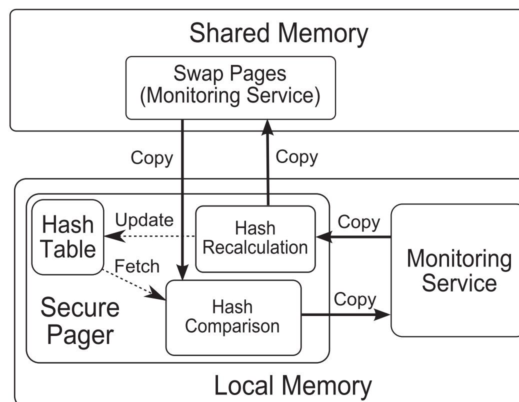


Fig. 5 Integrity checking.

3.3 Execution of Monitoring Service

The monitoring service can be executed as a binary file with the secure pager at kernel space. However, it needs to add many extra features to the secure pager if we want to manage the monitoring service, such as sleeping, exiting, or running multiple monitoring instances.

To solve this problem, we use a simple OS called xv6 [17] to execute the monitoring service as a user process. Xv6 is a simple Unix-like OS that provides some standard APIs and virtual memory functionality. With xv6, we can develop or optimize a monitoring service conveniently, and it is also easy to manage or execute multiple instances. Xv6, which is originally developed for the x86 architecture, is ported to running on SH-4A architecture for our implementation. After porting, xv6 can be divided into two parts: a kernel part (very small) and a RAM file system part (*fs.img*, needs to be much bigger) that contains the monitoring service and other files.

3.4 Automatic Update of Monitoring Service

We can extend our system freely by the automatic update function implemented in the secure pager, such as making optimizations or adding new features.

However, we intend to use Linux for loading the update file to shared memory, from where the secure pager can acquire it, thus

© 2013 Information Processing Society of Japan

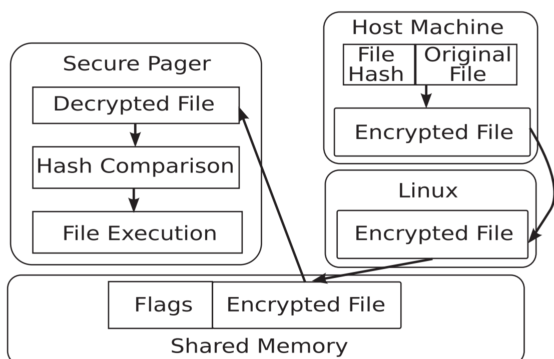


Fig. 6 Automatic update.

the update file may reveal information to Linux. To avoid this problem, we use the encryption method, and the decryption key is only stored in the secure pager. The update file is encrypted in the remote machine before Linux touches it, therefore Linux cannot get the decrypted content of the update file.

However, the encryption method only ensures that Linux that does not have the decryption key cannot get the decrypted content, but the encrypted content may be modified. The modified part can still be decrypted, but into meaningless data. In order to execute the correct update file, the integrity of the update file needs to be verified. Hash values of the update file, which are calculated before encryption, are used for validating the integrity during the update process. The hash values are prefixed to the update file to compose a new update file. With these steps, we can update the monitoring service in a secure manner.

A safe update procedure shown in **Fig. 6** is explained as following:

- Host machine: Hash values of every page of the update file are calculated in the host machine. The hash values are added as a prefix to the update file. Then the new update file is encrypted and copied into the Linux file system.
- Linux: The encrypted file is loaded into shared memory, and flags are set to show loading states.
- Secure pager: The prefixed part is firstly decrypted into the local memory to get the hash values. Then, the remaining part of the encrypted file is decrypted into the local memory in 1 or 2 pages at one time. The secure pager calculates hash values of these pages and compares them with the prefixed ones. If the hash values are equal, this part can be written to a new file in the xv6 RAM file system. Otherwise, the secure pager will ignore this update, delete the new file and continue to run the original monitoring service. This procedure is repeated with the integrity verification until the whole new update file is written into the file system. Then the update file will be executed to replace the original one or parallel with it. Since the update file can be divided into small parts for decryption, this method can also be applied to big monitoring services, even bigger than the local memory.

During the automatic update process, how the decryption key is stored is the key problem to keep security. The decryption key is stored in the remote host machine, and is inserted into the secure pager image when the host machine compiles the system boot image, which contains the secure pager image. We assume that the host machine, which can be well managed, and the RP1 firmware is trustworthy, hence Linux is unable to attack the host machine to get the key. As we mentioned in Section 3.1.1, RP1 can load the system boot image from the remote host machine via network. The decryption key is also stored in the system boot image when it is loaded into shared memory. This region will be swiped by the secure pager before Linux boots. During running, the key is only stored in the secure pager in the local memory, and Linux cannot access it. With these steps, Linux cannot get any information about the decryption key.

We can adopt two ways to update the monitoring service: single monitoring service pattern (S-pattern) and multiple monitoring services pattern (M-pattern). The S-pattern only runs one monitoring service in the system. Therefore, the space in the local memory used by the monitoring service is small, which is valuable due to the size of the local memory. However, the update introduces an interruption between ending the old monitoring service and starting the new one. The halt of the monitoring service may be a security hole in our system. On the other hand, the M-pattern executes multiple monitoring services at the same time, which occupy more space in the local memory or need more execution time for page swaps. The M-pattern is suitable for adding new functions to the old monitoring service without disturbing the monitoring functions. However, we have to replace the old monitoring service if there are some bugs that make it not viable for monitoring Linux. The ideal style could be the mixture of the S-pattern and the M-pattern. If the old monitoring service needs to be replaced, the S-pattern is used, and if only new functions need to be added into the system, it is preferred to choose the M-pattern.

However, because of the small local memory space in the development board, the S-pattern is chosen to update the monitoring service in our system. We will introduce system schemes in the following section.

3.5 System Architecture

Firstly, we implement the system architecture with an original scheme. We notice that some changes on the original scheme may improve performance. We call the modified one the optimized scheme. The details of these two schemes will be introduced in the following part.

3.5.1 Original Scheme

The original scheme is shown in **Fig. 7**. When the system starts, the secure pager is relocated into the local memory, and xv6 and Linux are relocated into shared memory. The secure pager firstly calculates hash values of all the pages of xv6, stores them into the hash table and runs xv6 to execute the monitoring service. After the monitoring service starts, Linux will start to provide applications for users. The monitoring service monitors the state of Linux, and its integrity is checked by the secure pager. At the same time, the secure pager checks the flags whether there needs to update the monitoring service. If the monitoring service is compromised, the secure pager can detect it and reboot to restore the whole system. With the integrity checking and the automatic update of the monitoring service, it can provide a reliable system with extensibility.

© 2013 Information Processing Society of Japan

3.5.2 Optimized Scheme

Based on the original scheme, we propose an optimized scheme with changes of the xv6 relocation shown in **Fig. 8**. As mentioned above, we know that xv6 is divided into a kernel part (small) and a RAM file system part (*fs.img*, comparable big). Therefore, we place the kernel part permanently in the local memory and only place the *fs.img* in shared memory. When the system boots, the secure pager and the xv6 kernel are loaded into the local memory, and the *fs.img* is relocated into shared memory. Then it jumps to run the secure pager. The secure pager only calculates hash values of the *fs.img*, and starts to run xv6. Page swaps occur only when some content stored in the *fs.img* needs to be used. Because the xv6 kernel would occupy part of the local memory, the space used for executing the monitoring service decreases. We consider that this method may reduce the overhead by lowering the number of page swaps between shared memory and the local memory. We will discuss more details in the next section.

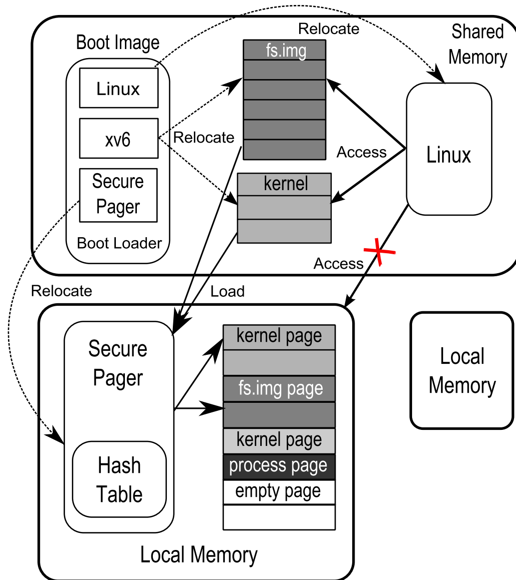


Fig. 7 Original scheme.

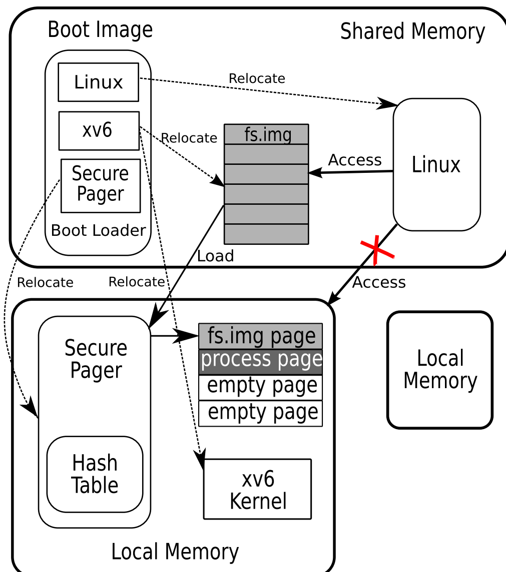


Fig. 8 Optimized scheme.

4. Evaluation

In this section, we will introduce system configurations for evaluation firstly. Then, functions of our system are verified. After this, decryption performance in the automatic update and overhead that may occur in the secure pager are analyzed. We also evaluate whether the secure pager causes influence to the performance of Linux. Finally, a conclusion is presented for this section.

4.1 System Configurations

In order to evaluate our system, we use 3 kinds of configurations shown in **Fig. 9**. In (A), Linux runs on 3 cores and xv6 runs on the other one core's local memory with the original scheme; In (B), Linux runs on 3 cores and xv6 runs in the other one core's local memory with the optimized scheme; In (C), Linux runs on 3 cores and xv6 runs on the other one core without the secure pager.

We choose a monitoring service, which can check the kernel system call table of Linux and the *hide_task* rootkit, and calculate execution time of the monitoring service to evaluate the overhead. The size of the monitoring service is less than 8 Kbytes, the size of the xv6 kernel is smaller than 64 Kbytes and the hash algorithm is SHA1.

We evaluated functions and performance of our system with these configurations.

4.2 Integrity Check Function

We ran the monitoring service in the configuration (A) and loaded a kernel module in Linux that could crack the content of xv6 stored in shared memory. When the modified content needed to be loaded into the local memory, the secure pager could detect these changes, print out error messages and reboot the system.

The secure pager can ensure the reliability of the monitoring service by checking the integrity when Linux runs. The reliability of the whole system is also improved.

4.3 Automatic Update Function

In order to validate the automatic update function, firstly we ran a monitoring service that could only check the kernel system

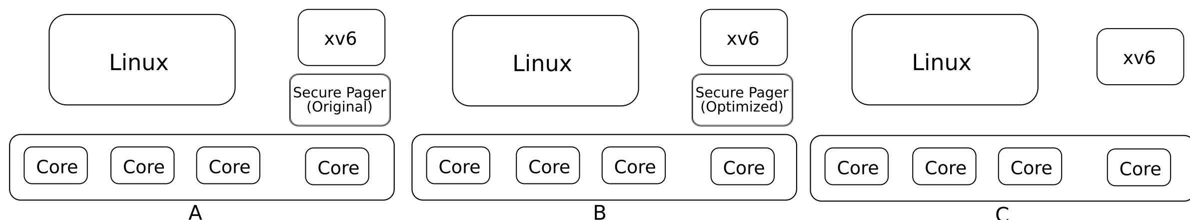


Fig. 9 System configuration.

© 2013 Information Processing Society of Japan

call table, and then we tried to update it to another one that added the function of checking the *hide_task* rootkit. We used a kernel module in Linux that could crack the content of the encrypted update file, and used another kernel module for hiding a task as malware.

When the first module was not loaded in our experiment, the monitoring service could be updated normally, and it could detect the *hide_task* rootkit that existed in Linux to show the successful update. When we did load this module in our experiment, the secure pager could detect the modification on the encrypted update file, print out error messages, reject the update and continue to execute the original monitoring service.

Since the decryption key is only stored in the secure pager and the prefixed hash values can be utilized to verify the integrity of the update file, the secure pager only allows the correct file to be updated. This function provides an easy and secure manner to update the monitoring service.

4.4 Decryption Performance of Update File

Since the encryption is able to be done in the host machine that has much better performance than embedded processors and users care nothing about the encryption time, we focused on the decryption performance in our system. We used 3 encryption mechanisms separately in the configuration (A): RSA, DES and a simple encryption mechanism, which only divides every page of the monitoring service into 2 parts and exchanges them. Although the simple mechanism may reveal messages about the update file, it is used as a sample to analyze the influence of the encryption method by comparing to other mechanisms. DES mechanism is less complex than RSA. However, both of them is popular in today's security applications. We encrypted the same monitoring service with these 3 mechanisms in the host machine and did the update in RP1. The time listed in **Table 1** shows the decryption performance.

According to Table 1, it shows that RSA takes much more time than DES and the simple encryption mechanism. This complex encryption mechanism may be unusable on embedded platforms for the limited performance of embedded processors. The performance of the simple encryption mechanism is very good but with the vulnerability of revealing information. The DES mechanism is a viable method in our platform. We should remind that this overhead only occurs when the monitoring service needs to be updated and does not influence the performance of the monitoring service during runtime. In real applications, a proper encryption mechanism is required to be chosen to fit the configuration of embedded platforms to obtain a good balance between security and performance.

4.5 Read Attacks Protection Overhead

We can use encryption methods on the part of xv6 stored in shared memory to avoid "read attacks." However, the encryption and the decryption of this part should cause overhead to the performance of the monitoring service and this overhead influences the monitoring service continually. According to Table 1, complex encryption methods are unsuitable for embedded systems. We chose an encryption mechanism, stream cipher RC4 [18], to evaluate the overhead in the configuration (A), and computed the execution time of the monitoring service with or without RC4 encryption.

The results shown in **Table 2** illustrate that this encryption mechanism introduces a reasonable overhead during the running of the monitoring service. The encryption mechanism should be applied depending on the hardware configuration to get a reasonable tradeoff between security and performance.

Table 1 Decryption time of update file.

Encryption Mechanism	Decryption Time
Simple	3.3 ms
DES	748.6 ms
RSA	More Than 25 Mins

4.6 Comparison between Original and Optimized Schemes

In order to compare the original scheme and the optimized scheme, we ran the same monitoring service in the configuration (A) and the configuration (B), and the execution time of the monitoring service was calculated shown in **Table 3**.

From Table 3, it indicates that the execution time in the configuration (B) is much shorter than in the configuration (A) and there are 22 page swaps in the configuration (A). We consider that the number of page swaps incurs this large difference. In the original scheme, both the *fs.img* (containing the monitoring service) and the xv6 kernel are stored in shared memory. When the monitoring service uses system calls, pages of the xv6 kernel are swapped in and out to run the monitoring service. Due to page copy and hash calculation during the page swap, the speed of the monitoring service in the original scheme is slow. However, in the optimized scheme, the xv6 kernel is located in the local memory permanently. Even if system calls occur, no pages need to be swapped, and the speed is not slowed as the original scheme.

Then, the overhead of the page copy and the hash calculation in our system were evaluated.

Firstly, we used a hardware method to copy pages between shared memory and the local memory with the data transfer unit (DTU). Compared to memory copy using software instructions, copying pages with DTU should be faster. After we applied both methods in the configuration (A), we obtained the execution time of the monitoring service in **Table 4** and the copy time of each page (22 pages) is 498 μs (software method) or 5 μs (DTU). It shows that the DTU method is much faster than the software

Table 2 Read attacks protection overhead (ms).

Encryption Mechanism	Execution Time
None	94.3
RC4	172.9

Table 3 Monitoring service evaluation.

Configuration	Execution Time (μs)	Page Swaps (times)
A	94,325.8	22
B	4,202.5	0

Table 4 Page copy overhead evaluation (μs).

Copy Type	Execution Time	Per Page (22 pages)
Software	94,325.8	498
DTU	83,743.8	5

© 2013 Information Processing Society of Japan

method, and the memory copy takes much time during the execution of the monitoring service.

Secondly, we used another hash algorithm MD5 in the secure pager with simpler complexity and less computing calculation than SHA1. We applied these two methods to the secure pager in the configuration (A) to get the execution time of the monitoring service shown in **Table 5** and the computing time of each page (22 pages) is 3,406.5 μs (SHA1) or 1,134.9 μs (MD5). We can find that the computing time with MD5 is much less than RSA, and the hash calculation occupies most execution time of the monitoring service. Although MD5 is not so robust in today's applications, the result shows that a proper hash mechanism needs to be selected to provide a good tradeoff between security and performance.

4.7 Performance of Linux

Finally, we verified whether the performance of Linux was influenced by the secure pager. We ran several test tools from Unixbench [10] in the configuration (A) and the configuration (C) to evaluate the performance of Linux. Three tools were chosen from Unixbench: Dhrystone 2 using registers variables (Dhry2reg), Double-Precision Whetstone (Whetstone) and System Call Overhead (Syscall). These three tools ran 10 times with a single task and 5 times with 4 parallel tasks. The average scores are shown in **Fig. 10**, **Fig. 11**, **Fig. 12**.

Table 5 Hash calculation overhead evaluation (μs).

Hash Algorithm	Execution Time	Per Page (22 pages)
SHA1	83,743.8	3,406.5
MD5	33,782.4	1,134.9

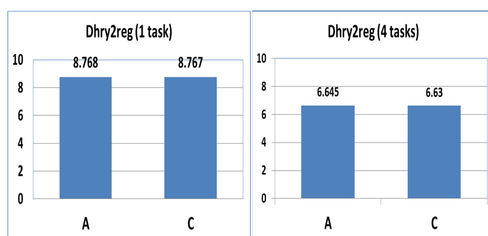


Fig. 10 Dhry2reg results (10^6 *loops*).

Fig. 11 Whetstone results (*MWIPS*).

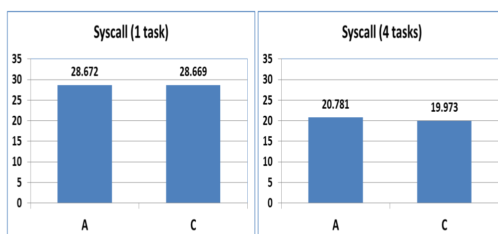


Fig. 12 Syscall results (10^5 *loops*).

From these figures, we can argue that the performance of Linux is minimally influenced by the secure pager. We consider that the influence is from the passive monitoring service, which would occupy memory bus during runtime. The secure pager may increase the execution time of the monitoring service, which may lead to more bus time occupation. However, because the monitoring service does not use a core shared with Linux, it does not block the running of Linux. As the results show, the secure pager have minimal influence on the performance of Linux.

We have evaluated and analyzed some factors that may influence the performance. To conclusion, the secure pager brings very little overhead to Linux in our architecture, and it provides functions of the integrity checking and the automatic update of the monitoring service. The encryption mechanism, the page copy method and the hash algorithm are required to be chosen carefully to meet the performance of embedded platforms.

We should notice that using one core of the processor for secure purpose in this architecture may lead to resource waste in embedded systems. However, we can foresee that using a special core for enhancing the reliability would be reasonable consumption when the number of cores in one processor increases to 16 or more. On the other hand, there is a type of processors called heterogeneous multi-core architecture processors [15], [30], in which each core can have its own configuration specially for different workloads to provide a good balance between performance and resource consumption. We can use a single core with a considerable configuration to run the secure pager and the monitoring service, and obtain a reasonable tradeoff between resource consumption and security. Depending on processors with a proper architecture, our architecture can provide a robust secure system without occupying overmuch resource.

5. Conclusion

In this paper, we proposed an extensible OS architecture that can provide enhanced reliability to the whole system without causing heavy overhead. In this architecture, isolation between the monitoring service and the guest OS is provided based on the local memory, a hardware feature, without the dependency of a hypervisor. The monitoring service can avoid attacks from the compromised guest OS due to the isolation. A secure pager is adopted to generalize this architecture by extending the size of the local memory virtually, and extend monitoring functions by the automatic update of the monitoring service. The evaluations demonstrate that we could use this architecture to build an extensible secure system with reasonable resource consumption on processors with many cores or heterogeneous architecture equipped with local memory.

6. Future Work

We will focus on the followings for future research:

- Secure pager and xv6 need more optimizations to reduce the memory occupation and improve the performance.
- More experiments need to be conducted to find how to determine a suitable swap mechanism depending on the size of the local memory.
- Apply this method on other types of processors.

© 2013 Information Processing Society of Japan

References

[1] ARM: TrustZone, ARM Cooperation (online), available from ⟨http://www.arm.com/products/processors/technologies/trustzone.php⟩ (accessed 2013-06-25).

[2] Armand, F. and Gien, M.: A Practical Look at Micro-Kernels and Virtual Machine Monitors, *Proc. 6th IEEE Conference on Consumer Communications and Networking Conference*, Las Vegas, Nevada, pp.1–7 (2009).

[3] Azab, A.M., Ning, P., Wang, Z., Jiang, X.X., Zhang, X.L. and Skalsky, N.C.: HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity, *Proc. 17th ACM Conference on Computer and Communications Security*, Chicago, IL, pp.38–49 (2009).

[4] Baron, M.: Single-Chip Cloud Computer, Intel (online), available from ⟨http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-article.html?wapkw=single-chip+cloud+computer⟩ (accessed 2013-06-25).

[5] CERT: CERT/CC Security Improvement Modules: Detecting Signs of Intrusion, Technical Report sei-sim-001, CERT Coordination Center (1997).

[6] CVE: CVE list, CVE (online), available from ⟨http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Xen⟩ (accessed 2013-06-25).

[7] Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A. and Chen, P.M.: ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay, *Proc. 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, pp.211–224 (2002).

[8] Garfinkel, T. and Rosenblum, M.: A virtual machine introspection based architecture for intrusion detections, *Proc. 10th Network and Distributed Systems Security Symposium*, Diego, California, pp.191–206 (2003).

[9] Garfinkel, T. and Rosenblum, M.: When virtual is harder than real: Security challenges in virtual machine based computing environments, *Proc. 10th Conference on Hot Topics in Operating Systems*, Santa Fe, New Mexico, pp.74–82 (2005).

[10] Google: Byte-unixbench: A Unix benchmark suite, Google Project (online), available from ⟨http://code.google.com/p/byte-unixbench/⟩ (accessed 2013-06-25).

[11] Hay, B. and Nance, K.: Forensics Examination of Volatile System Data Using Virtual Introspection, *ACM SIGOPS Operating Systems Review*, Vol.42, No.3, pp.20–25 (2008).

[12] Jones, S.T., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid, *Proc. 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, WA, pp.91–100 (2008).

[13] Kinebuchi, Y.: Software and Hardware Supports for Multi-OS Environment, PhD Thesis, Waseda University, Tokyo (2012).

[14] Kinebuchi, Y., Butt, S., Ganapathy, V., Iftode, L. and Nakajima, T.: Monitoring Integrity using Limited Local Memory, *IEEE Trans. Information Forensics and Security*, Vol.8, No.7, pp.1230–1242 (2013).

[15] Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P. and Tullsen, D.M.: Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction, *Proc. 36th International Symposium on Microarchitecture*, San Diego, CA, pp.81–92 (2003).

[16] Li, C.X., Raghunathan, A. and Jha, N.K.: Secure Virtual Machine Execution under an Untrusted Management OS, *Proc. 3rd IEEE International Conference on Cloud Computing*, Miami, Florida, pp.172–179 (2010).

[17] MIT: Xv6, a simple Unix-like teaching operating system, Massachusetts Institute of Technology (online), available from ⟨http://pdos.csail.mit.edu/6.828/2012/xv6.html⟩ (accessed 2013-06-25).

[18] Mousa, A. and Hamad, A.: Evaluation of the RC4 Algorithm for Data Encryption, *International Journal of Computer Science & Applications*, Vol.3, No.2, pp.44–56 (2006).

[19] Payne, B.D., Carbone, M. and Lee, W.: Secure and Flexible Monitoring of Virtual Machines, *Proc. 12th Computer Security Applications Conference*, Seoul, Korea, pp.385–397 (2007).

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

[21] Shimada, H., Courbot, A., Kinebuchi, Y. and Nakajima, T.: A Lightweight Monitoring Service for Multi-core Embedded Systems, *Proc. 13th IEEE International Symposium on Object Component Service-Oriented Real-Time Distributed Computing*, Carmona, Seville, Spain, pp.202–209 (2010).

[22] Shimizu, K., Nusser, S., Plouffe, W., Zbarsky, V., Sakamoto, M. and Murase, M.: Cell Broadband Engine^{TM} processor security architecture and digital content protection, *Proc. 4th ACM International Workshop on Contents Protection and Security*, Santa Barbara, CA, pp.13–17 (2006).

[23] Srivastava, A., Singh, K. and Giffin, J.: Secure Observation of Kernel Behavior, Technical Report gt-cs-08-01, Georgia Institute of Technology (2008).

[24] Szefer, J. and Lee, R.B.: Architectural Support for Hypervisor-Secure Virtualization, *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, pp.437–449 (2012).

[25] VMware: Performance Evaluation of AMD RVI Hardware Assist, VMware, Inc. (online), available from ⟨http://www.vmware.com/pdf/RVI_performance.pdf⟩ (accessed 2013-06-25).

[26] VMware: Performance Evaluation of Intel EPT Hardware Assist, VMware, Inc. (online), available from ⟨http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf⟩ (accessed 2013-06-25).

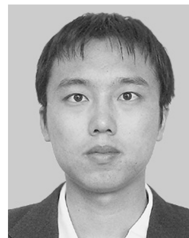
[27] Wagner, D. and Schneier, B.: Analysis of the SSL 3.0 Protocol, *Proc. 2nd USENIX Workshop on Electronic Commerce*, Oakland, California, pp.29–40 (1996).

[28] Wang, X.Y., Feng, D.G., Lai, X.J. and Yu, H.B.: Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD, Technical Report 2004/199, Cryptology ePrint Archive (2004).

[29] Wang, Z. and Jiang, X.X.: HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity, *Proc. 31th IEEE Symposium on Security and Privacy*, Berkeley, CA, pp.380–395 (2010).

[30] Wei, T.Y., Qiu, Z.L., Young, C.P. and Chang, D.W.: Development of Heterogeneous Multi-core Embedded Platform for Automotive Applications, *Proc. International Conference on Circuits, System and Simulation*, Bangkok, Thailand, pp.193–197 (2011).

[31] Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K. and Kasahara, H.: A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption, *Proc. Solid-State Circuits Conference*, San Francisco, CA, pp.100–101 (2007).



Ning Li was born in 1984. He received his M.S. degree from Shandong University, China, and is a Ph.D. candidate in the Department of Computer Science and Engineering in Waseda University, Tokyo, Japan. Now he receives the scholarship from China Scholarship Council (CSC). His research interests are operating systems, security, system architecture and embedded systems.

Yuki Kinebuchi obtained his Ph.D. in Computer Science and Engineering at Waseda University. His research interests are operating systems, system virtualization, security, system architectures, and embedded systems.

Hiromasa Shimada is a Ph.D. candidate in the Department of Computer Science and Engineering in Waseda University, Tokyo, Japan. His research interests are operating systems, security and system architecture.

© 2013 Information Processing Society of Japan

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 during 1990–1993, a research engineer in AT&T Laboratories, Cambridge during 1998–1999 and a visit 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 during 1993–1999. He was program co-chair of RTCSA 2002 and ISORC 2003, and general chair of RTCSA 2003 and ISORC 2005. His research interests include distributed systems, operating systems, ubiquitous computing and information appliances.

© 2013 Information Processing Society of Japan