A Local-Memory-Based Extensible Secure Architecture for Embedded Systems

組込みシステムのためのローカルメモリを用いたセキュア・アーキテクチャ

February 2014

WASEDA UNIVERSITY
GRADUATE SCHOOL OF FUNDAMENTAL SCIENCE AND ENGINEERING
MAJOR IN COMPUTER SCIENCE AND ENGINEERING
RESEARCH ON DISTRIBUTED SYSTEMS
DOCTOR OF ENGINEERING

Ning LI

© Copyright by Ning LI 2014
All rights reserved

To my parents:

I would like to thank them for bringing me up and supporting me every step of the life. They always respect my choice, and their love and help enabled me to face up to any difficulties and make me the person I am today. Without their words, I could never have walked this far. All wishes to them.

Acknowledgements

First of all, I would like to acknowledge professor Tatsuo Nakajima, my supervisor, for introducing me to the fantastic world of computer science research, continually guiding me and giving me confidence and knowledge on this road. Without his help, I will never go to this far.

I would also like to acknowledge professor Shigeki Goto, Professor Hayato Yamana, Professor Hiroshi Yamada, my defense's reviewers, for their help to complete this dissertation and improve it to such a high level.

Thanks to Weihe, Li and Shuhua Men, my parents, for their support during my doctor's study. Without their love, it is impossible for me to stand here.

I would like to acknowledge Chao Che, a very very nice girl, for her so much help and support to plan a bright future and giving me many hints during the writing of the dissertation.

Thanks to Yefeng Liu, Hiromasa Shimada, Tsung-Han Lin, Han Chen, Connsynn Chye, Hairihan Tong et al., my lab's members, for their help during my doctor's research and a happy research life with them.

Last but not least, I would like to acknowledge Jing Li, Wei Gao, Zhen Yan, Ming Chen, Gangfeng Liu, Yanwei Li et al., all my friends, for their encouragement and support.

All best wishes to them.

Abstract

Embedded hypervisors can execute an integrity checker to monitor the state of a target OS and can isolate this integrity checker to enhance security. However, they add overhead to the target OS, and their own vulnerabilities may compromise the isolation, which undermines the security efforts. A machine architecture called Limited-Local-Memory (LLM) is proposed to implement a similar architecture on a multi-core processor using a hardware-centric method to isolate the integrity checker. It has some characteristics suitable for embedded systems, such as low overhead, a small trusted code and minimal modifications to the target OS. However, in current research, the LLM architecture has only been emulated in the virtual machine monitor, NOT in real platforms, and it somewhat assumed a large fraction of the local memory. It has rarely been equipped on embedded processors.

In this paper, we propose a secure OS architecture based on the LLM architecture for embedded systems and implement it on a real embedded platform. We also extend the traditional LLM architecture by automatically updating the integrity checker to enhance the security functions without disturbing the running of the target OS. Furthermore, we apply many methods to reduce the demand on the local memory and propose some hardware recommendations for the LLM architecture. Our research illustrates the efficiency of the LLM architecture and generalizes its application for embedded systems by reducing the local memory requirement. In this work, we propose an extensible secure architecture for embedded systems with low overhead, which consumes a reasonable amount of resources on appropriately configured processors.

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

2 Related Work **9**

2.1 Introduction to Hypervisors 9

2.2 Security Approaches based on Hypervisors and Peripheral Devices 15

2.3 LLM Machine Architecture 18

3 A Local-Memory-Based Extensible Secure Architecture for Embedded Systems **23**

3.1 RP1 and SuperH Processor Architecture 23

3.2 Methods 26

3.2.1 System Architecture 26

3.2.2 Secure Pager 27

3.2.3 Reduce Size of Page Tables of the Monitor OS 31

3.2.4 Extensibility 33

3.3 Threat Model 33

3.3.1 Assumption 33

3.3.2 Security of the Target OS 35

3.3.3 Security of the Secure Pager 37

3.3.4 Security of the Monitor OS 37

3.4 Implementation 38

3.4.1 Implementation Environment 39

3.4.1.1 Boot Mechanism 42

3.4.2 Integrity Verification of Integrity Checker 42

3.4.3 Execution of Integrity Checker 44

3.4.4 Automatic Update of Integrity Checker 45

3.4.5 System Architecture 48

3.4.5.1 Original Scheme 48

3.4.5.2 Optimized Scheme 49

3.5 Evaluation 51

3.5.1 System Configuration 51

3.5.2 Performance of Linux 53

3.5.3 Integrity Check Function 55

3.5.4 Automatic Update Function 55

3.5.5 Decryption Performance of Update File 56

3.5.6 Read Attacks Protection Overhead 57

3.5.7 Comparison between Original and Optimized Schemes . . 58

3.5.8 Future Analysis of the Influence of the Secure Pager 62

4 Discussion **77**

4.1 Hardware Virtualization Support in Embedded Processors 77

4.2 TLB Management 79

4.3 Cache Management 80

4.4 Hardware Features in Our Approach 83
4.5 Discussion and Hardware Recommendation 84

5 Conclusion **87**

6 Future Work **89**

References **91**

7 Publication List **103**

List of Figures

1.1 Detection tools in host OS. 3

1.2 Detection tools in hypervisors. 4

1.3 Detection tools in LLM machine architecture. 6

2.1 Secure computing environment based on hypervisors. 10

2.2 Hypervisors on a host operating system. 13

2.3 Hypervisors on rare hardware. 13

2.4 Xen hypervisor architecture. 14

2.5 Detection based on peripheral devices. 17

2.6 LLM machine architecture. 19

3.1 RP1 board. 24

3.2 Memory architecture in SuperH. 25

3.3 System architecture. 27

3.4 Work mechanism of secure pager. 28

3.5 Execution address for the monitor OS. 30

3.6 Page table format of the monitor OS. 30

3.7 Two-level page table in x86 architecture. 32

3.8 Internal scheme of secure pager. 34

3.9 Integrity checking. 44

3.10 Automatic update. 46
3.11 Original scheme. 49
3.12 Optimized scheme. 50
3.13 System configuration. 52
3.14 Dhry2reg results (10^6 loops). 53
3.15 Whetstone results (MWIPS). 53
3.16 Syscall results (10^5 loops). 54
3.17 Dhry2reg results (10^6 loops). 54
3.18 Whetstone results (MWIPS). 54
3.19 Syscall results (10^5 loops). 55
3.20 Split large integrity checkers. 60
3.21 *kernel.o* is located permanently in local memory in configuration (A). 63
3.22 Add *proc.o* into local memory. 65
3.23 Add *console.o* into local memory. 67
3.24 Add *console.o* and *scif.o* into local memory. 68
3.25 A bad case of relocation of the kernel. 70

4.1 Recommended hardware architecture. 86

6.1 Extended system architecture. 90

List of Tables

2.1 Size of hypervisors (KLOC). 12

3.1 Page table size comparison (Bytes). 33

3.2 LOC of files in the secure pager. 41

3.3 Code size in Kinebuchi's research. 42

3.4 Decryption time of update file. 56

3.5 Read attack protection overhead (ms). 57

3.6 Integrity checker evaluation. 58

3.7 Page copy overhead evaluation (ms). 59

3.8 Hash calculation overhead evaluation (ms). 59

3.9 Total execution time of the large integrity checker in configuration (A). 61

3.10 Total execution time of the large integrity checker in configuration (B). 61

3.11 Single instance execution time in configuration (A). 61

3.12 Single instance execution time in configuration (B). 62

3.13 Total execution time of the large integrity checker with *kernel.o* located in local memory in configuration (A). 64

3.14 Single instance execution time with *proc.o* located in local memory. 64

3.15 Total execution time of the large integrity checker with *proc.o* located in local memory. 66

3.16 Single instance execution time with *kernel.o* located in local memory. 66

3.17 Total execution time of the large integrity checker with *console.o* or *console.o* + *scif.o* located in local memory. 66

3.18 Single instance execution time with *console.o* or *console.o* + *scif.o* located in local memory. 69

3.19 Total execution time of the large integrity checker with a bad case. 69

3.20 Single instance execution time with a bad case. 71

3.21 Execution time in configuration (A) with reduction of spare space. 72

3.22 Execution time in configuration (A) with or without a *sleep()* function. 73

3.23 Execution times of functions in configuration (A) (sample). 74

1

Introduction

Computing environments are now part of many aspects of life, including mobile devices (phones, PDAs), desktop computers, notebooks, workstations and servers. Computing environments can offer people great convenience with a rapid computing speed that is much higher than the same computation in the brain. However, as people depend increasingly heavily on computing devices , the security of computing devices is steadily becoming a more critical issue. No one wants to reveal private information to those whom he or she does not trust, especially in businesses. At the same time, attackers always seek to acquire such informations, and the potential benefits are often large enough to inspire many threats. Once network connections are introduced to personal mobile devices, the situation becomes even worse. An attacker does not need to obtain your physical device, break your password or other security measures and then obtain your sensitive information. Instead, an attacker can access your computing devices from a distance via the network to try to obtain your data. We cannot expect end users to have sufficient knowledge of computing technologies to protect against such attacks, therefore, enhancing the security of the computing device becomes an important part of system development.

A computing device includes hardware and software, both of which may be attacked. Hardware security replies mainly on manufacturers, and if they produce vulnerable devices, the security of the software developed to run upon them is irrelevant: the entire device can be easily attacked by hardware attackers to

obtain control of the whole device, including the hardware and the software. The software may be unsuitable to detect these attacks for existing hardware limitations. We consider the hardware to be the foundation of the software security. Sometimes the software can help to identify the vulnerabilities in the hardware. However, software is more likely to rely on the hardware for security.

The software in computing devices is also vulnerable to attackers. Compared to hardware vulnerabilities that may require hardware modifications to exploit, software vulnerabilities can be attacked with little effort on the part of the attacker. In fact, many attacking tools can be easily downloaded from the network, and anyone with basic language can exploit them without great difficulty. During a system lifetime, there are two critical factors in the security of the software: 1) how to boot software in a secure manner and 2) how to ensure its security during runtime. Trusted boot methods are proposed to solve the first problem. These methods are usually based on a hardware unit called a Trusted Platform Module (TPM) [2] [4] [8] [11] [24] [40] [42] [68]. A TPM is bonded to a unique hardware platform and can verify the boot process of the platform. It can also produce quotes for verification by third parties. Based on TPM technologies, Intel and AMD have developed their own trusted boot technologies, the Trusted eXecution Technology (TXT) [4] [8] and Secure Virtual Machine (SVM), respectively, which can securely boot up a process (late launch) during runtime [40]. Even if other components of the system are malicious, these hardware mechanisms are still guaranteed to boot the correct process. Some studies have further developed the TPM technologies to virtualize the TPM and thus provide the trusted boot functionality to multiple virtual machine instances that are intended to run on hypervisors. Security upon booting up a system can help ensure that we can access clean systems without malware.

After the system boot is completed, the system needs to use many kinds of data that we input or receive from other interfaces. The operations that we input may not be appropriate or attackers may try to hack the system. To protect OSs, detection tools are imported into the OSs, as shown in Fig. 1.1. These detection tools monitor critical behaviors of the system and judge whether these behaviors are trustworthy. Once they detect abnormal behaviors, they

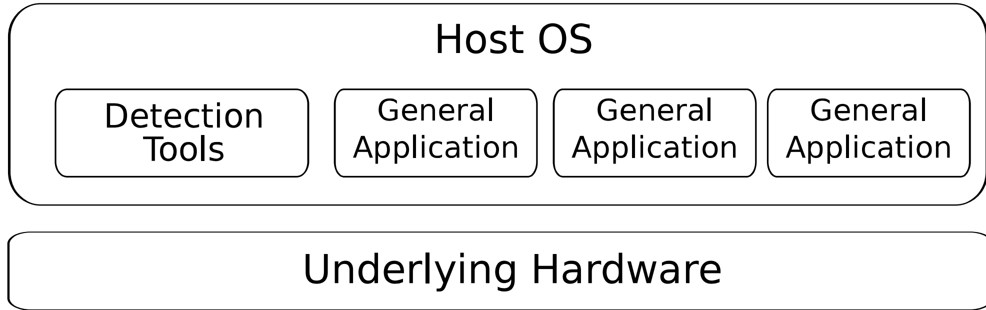

Figure 1.1: Detection tools in host OS.

will inform users to deal with them or perform recoveries to ensure that the system is running in a normal state. However, the detection tools cannot detect all types of malware, and they must be updated to detect the newest malware. There is a gap between the creation of new malware and applying detection technologies to identify it, which allows the malware to attack the system during certain periods and gives it the chance to infect the system kernel, which the detection tools rely on for monitoring tasks, to invalidate the detection tools. Once the system kernel is infected or the detection tools are invalidated, even if the detection tools are updated to detect the newest malware, the infected system might not execute the detection tools correctly. The user must then intervene to restore the system to a clean state, which is somewhat difficult for most end users. However, modern operating systems (OSs) may contains millions of lines of codes, and the tests before releasing are often insufficient to ensure that the code is completely trustworthy. This great amount of code broadens the system's vulnerability to attackers. The dependency on the system kernel makes it difficult for detection tools to protect the system kernel against malware that exploits unknown vulnerabilities.

Removing the dependency of the detection tools on the system kernel can efficiently solve this problem. Many studies [23] [25] [29] [32] [52] [75] employ hypervisors to accomplish this task, as shown in Fig. 1.2. Hypervisors are software approaches that provide an abstract layer of underlying hardware for multiple OSs running concurrently on a single hardware device. Hypervisors have many advantages, such as high hardware resource usage, cost reduction, workbench migration, legacy software support and security. Hypervisors directly control the

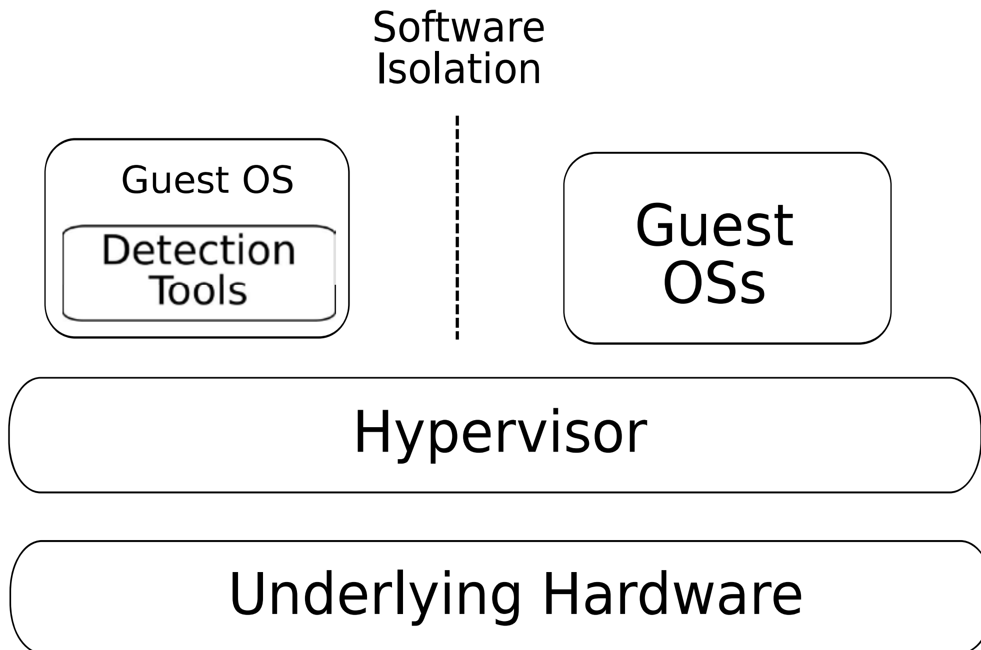

Figure 1.2: Detection tools in hypervisors.

necessary access to hardware resources (e.g., memory, I/O) so that any running OS can only access the permitted hardware resources and cannot touch other OSs if the hypervisors do not allow it. This isolation provided by the hypervisors can be used to move the detection tools out of the system and thus remove the dependency on the system kernel. Then, the detection tools can check the state of the system without falling victim to infection. However, this method also introduces semantic barriers into the detection tools. They must acquire some knowledge about the target systems, such as its data structure or binary allocation, to obtain the necessary information to undergo checking tasks, and they depend on the hypervisors to acquire this information correctly.

Although hypervisors are designed to be more privileged than guest OSs and to be a thin software solution with good performance, they are not designed specifically for security. Furthermore, as hypervisors are equipped with more functions, the hypervisors become very large, thereby broadening the surface that is vulnerable to attackers. Many studies [6] [26] have shown that modern hypervisors, such as Xen and KVM, already contain many vulnerabilities in modern. It is foreseen that the situation may be worse in enterprise hypervisors, as a company

will always try to add more functions than its competitors to attract customers to use its service, and new functions usually mean more complex code. Many studies [22] [39] [58] [61] [66] have tried to design a small hypervisor. The basic ideas in these approaches are splitting the hypervisor into pieces and granting root privilege only to necessary components. Therefore, we would only need to ensure that these root-level components would be secure enough for the correct running of the hypervisors. However, some studies have shown that these components must still interact with other untrustworthy code in the hypervisors, and these operations may be exploited to compromise the root trust of the hypervisors [7] [9]. However, instead of continually reducing the size of hypervisors for the security purposes, it may be better to remove the hypervisor from the security chain and use other technologies to isolate the detection tools from the target system.

In embedded fields, microkernels and embedded hypervisors can also isolate detection tools from the target OS. Unfortunately, microkernels usually cause heavy performance overhead and require substantial modifications to the target OS. However, embedded hypervisors are not always equipped with isolation between guest OSs in performance-critical cases. Embedded hypervisors can also provide isolation functionality through the same method used by traditional hypervisors, but with substantial overhead. More methods are still needed to improve the performance of embedded hypervisors, such as importing hardware virtualization extensions or applying a dedicated method based on a special processor architecture. Obviously, embedded hypervisors themselves also face similar vulnerabilities as do traditional hypervisors.

Although it was not specially designed for embedded systems, Kinebuchi et al. proposed a machine architecture called Limited Local Memory (LLM) [34],which uses with a hardware-centric method, as shown in Fig. 1.3, to provide isolation, which can efficiently protect the detection tools and only depends on small codes to be trustworthy. The LLM architecture assumes a small private memory area for respective core and provides a tamper-proof execution environment for a rootkit detector, which was built as part of the study, to monitor the target OS without becoming infected. The LLM architecture provides some appropri-

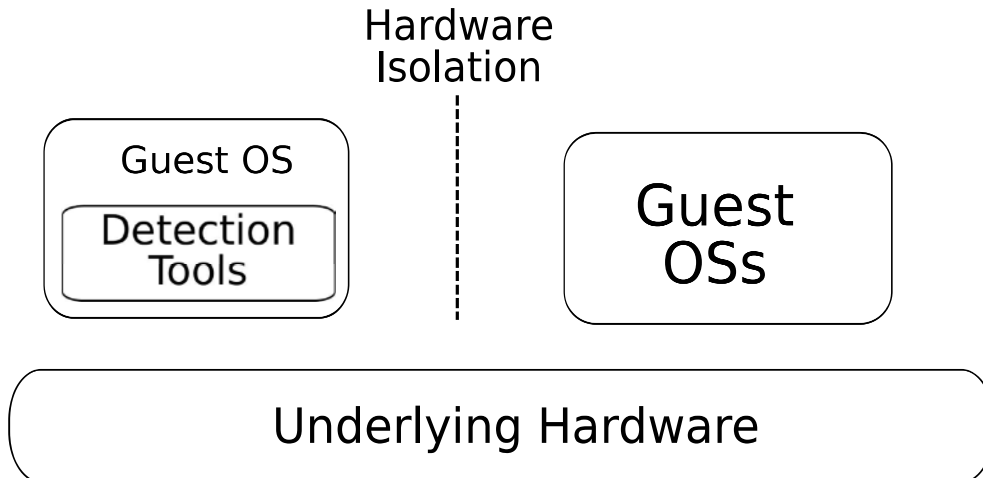

Figure 1.3: Detection tools in LLM machine architecture.

ate characteristics for embedded systems, such as isolation with low overhead, a small Trusted code and minimal modifications to the target OS. We consider the LLM architecture to be suitable for embedded systems with a security-specific purpose. However, the implementation in Kinebuchi's paper is emulated in a virtual machine monitor (VMM) QEMU that virtualizes the x86 architecture and assumes a local memory of 548 Kbytes, which is limited but moderately big. It is difficult to consider that there is so much private memory space for cores in embedded processors. For example, RP1 [76] and Cell B.E. [60] have a local-memory-similar architecture with respective sizes of 128 Kbytes and 256 Kbytes. Therefore, applying this method to a real embedded platform would be a well-founded verification of this method and show the efficiency of the LLM machine architecture, even when the local memory is very small.

In this paper, we apply this method to a real embedded platform with a LLM-similar hardware configuration that is equipped with a much smaller local memory. We build a secure architecture that can efficiently enhance the security of the embedded systems and extend the security functions in the architecture during runtime to protect against the newest vulnerabilities found in the target OS. This architecture requires only minimal modification to the target OS and causes low overhead. Furthermore, some hardware recommendations are presented to make the LLM architecture more applicable to real embedded processors. Our research

can illustrate the efficiency of the LLM architecture and generalize its application to embedded systems by reducing the local memory requirement.

In summary, the contributions of this paper are as follows:

- Implementation of an integrity checking system on a real embedded hardware with an LLM-similar architecture that is based on a small amount of local memory. This implementation demonstrates the viability of the LLM architecture for application on a real-world embedded processor to reduce the local memory requirements.

- Detailed design methods within a real embedded platform. The RP1 board is equipped with one SH-4A quad-cores processor with software-managed TLB, which can help to reduce the required size of the local memory. Therefore, the design details are significantly different from the design on a x86 platform with hardware-managed TLB.

- Building an extensible secure architecture that can efficiently enhance the security of embedded systems. We can securely update the security functions in the architecture without disturbing the normal running of the target OS to fix bugs, make optimizations or add new functions for the latest security vulnerabilities.

- Hardware recommendations for the LLM machine architecture. Based on the evaluation, we propose some recommendations to optimize the LLM architecture. We consider that these recommendations can help simplify and generalize the application of the LLM architecture.

The remainder of my dissertation is organized as follows. Section II introduces both studies for improving hypervisor security and the LLM machine architecture, and Section III describes how to build an extensible secure architecture on a real embedded platform RP1 and evaluate the system to show the performance and security of this architecture. In Section IV, I survey the hardware features used in our design, and discuss the possibilities of using them in future embedded systems. Section V draws some conclusions of the overall research, discusses our

approach and proposes some hardware recommendations for the LLM machine architecture. Section VI discusses future directions of our research.

2

Related Work

In this chapter, we will introduce work related to our research. Hypervisors can provide a similar security functionality to our research with different mechanisms. Therefore, we first introduce hypervisors and how they achieve a secure environment for running different OSs. Then, we introduce some approaches that have been designed to improve the security of hypervisors, followed by two more similar architectures, i.e., co-processor-based detection and the LLM machine architecture. From those approaches, we proceed to the research reported in this dissertation.

2.1 Introduction to Hypervisors

Since virtualization technology was proposed in the 1960s by IBM [41] [49], researchers have continued to add new functions to and optimize hypervisors. Currently, there are many modern enterprise hypervisors that supply us for many applications, such as cloud computing, workbench migration, legacy software support and cost reduction. Hypervisors can be divided into two types: full-virtualization and para-virtualization. Full-virtualization hypervisors can run unmodified operating systems, but adding noticeable overhead to the performance of guest operating systems. Conversely, para-virtualization hypervisors require modifications to the guest operating systems, but it is possible to opti-

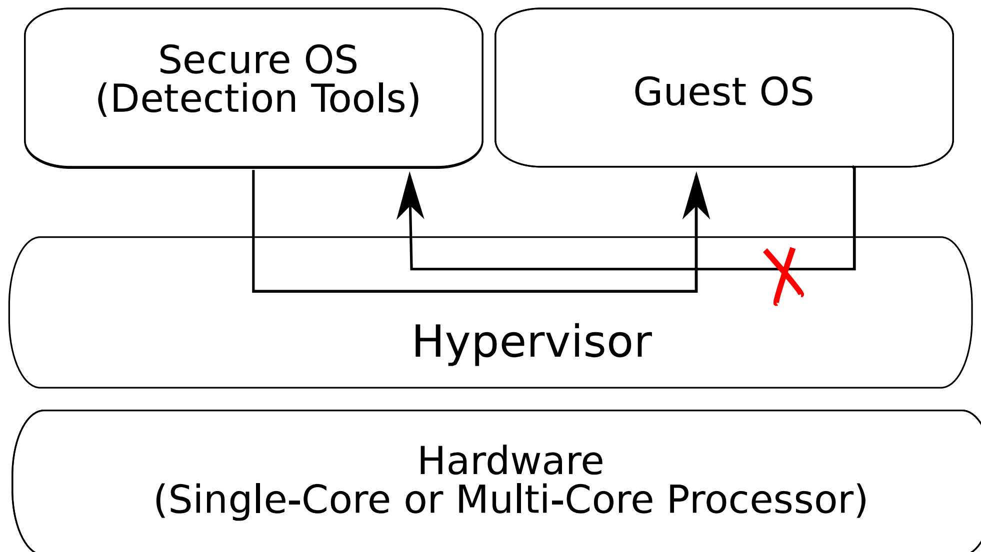

Figure 2.1: Secure computing environment based on hypervisors.

mize the guest OSs to achieve better performance when running on a hypervisor. Although security is not the primary purpose of hypervisors, they can control access to hardware resources by guest operating systems to ensure that guest operating systems cannot disturb one another's normal running. This characteristic has been applied in many works to build a secure computing environment based on hypervisors [23] [25] [29] [32] [52] [75], as shown in Fig. 2.1. We briefly introduce some details regarding these studies.

George W. Dunlap, Samuel T. King, Sukru Cinar et al. presented a system logger called Revirt [23]. Revirt uses UMLinux as the virtual machine to provide VMM to run guest OSs. The logger runs on the virtual machine, independent of the guest operating system and can log sufficient information to replay and analyze attacks that consist of non-deterministic events. The loggers incur minimal overhead, but the virtualization adds significant overhead to the kernel-intensive workload.

Liveware [25] is a VMI IDS (Virtual Machine Introspection Intrusion Detection System) proposed by Garfinkel and Mendel Rosenblum. This system is implemented by adding a hook to VMware and provides three useful properties to the VMI IDS: isolation, inspection, and interposition. Using the interface library

and policy engine presented in the paper, Liveware provides functions to interact with the VMM and a host OS, detect VMM and the host OS status and respond in an appropriate manner. However, it adds heavy overhead to the system.

Brian Hay and Kara Nance presented a suite of virtual introspection tools developed for Xen (VIX tools) [29]. This VIX tool suite can be used for the digital forensic examination of volatile system data in virtual machines. This VMM requires many steps to translate the virtual address of Linux to an address space that can be accessed by the trusted OS. With the address translation, the VIX tool suite can acquire volatile system data to monitor the state of Linux. However, this system would be improved by providing support for more operating system types, such as Windows, and making the virtual introspection techniques suitable to other virtualization platforms, such as VMware.

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau proposed Lycosid: a VMM-based hidden process detection and identification system [32]. This system contains two views for detection: one is a trusted view achieved from within a VMM using a tool called Antfarm, and the other is an untrusted view obtained from the guest OS using the *ps* (Linux) or *pslist.exe* or *tasklist.exe* (Windows) commands. This system uses mathematical methods to judge whether there are one or several hidden processes and to identify the likely number of hidden processes. However, as the number of hidden processes increases, the system requires more time to determine and identify the number of the hidden processes.

Lares [52] is a VMM tool that features the ability to perform active monitoring while still benefiting from the increased security of an isolated virtual machine. In this paper, the architecture and a prototype implementation are discussed. In Lares, the security VM can actively monitor the guest OS via a hook in the guest OS, which is Windows XP running on Xen. The security and performance of Lares are evaluated. The evaluation demonstrates that Lares has a low impact on the system performance. However, malware may attack the hook in the guest VM. Although there is security protection on the hooks, more effective protection of the integrity of the hook remains a problem.

Table 2.1: Size of hypervisors (KLOC).

Hypervisor	Code Size
Xen	100 (+200, trusted VM)
KVM	220
ESXi	200
Hyper-V	100 (+more than 400, Windows Server 2008)

The μDenali [75] was designed, implemented and evaluated by Andrew Whitaker, Richard S. Cox, Marianne Shaw et al. It is an extensible and programmable virtual machine monitor that can run modern operating systems. It allows a programmer to modify the virtual architecture to easily implement new virtual services, such as manipulating disks or capturing and migrating virtual machine states. They also presented an application-level API to simplify writing the extensions used to describe and evaluate the system. While μDenali is a flexible virtual machine monitor that can easily add new virtual services, it also adds significant overhead to the system.

Hypervisors can run in the host operating system or directly in rare hardware, as shown in Fig. 2.2 and Fig. 2.3. Xen is unusual in that it requires a trusted VM to manage the other VMs, as shown in Fig. 2.4. Hypervisors are assumed to be thinner and more secure than the guest operating systems running upon them, so that we should ordinarily trust hypervisors (including the trusted VM in Xen) to control all access to the underlying hardware. However, modern hypervisors are becoming complex, with a large code size, especially in enterprise hypervisor products that include more functions to attract customers. The increasing code size broadens the surface of modern hypervisors vulnerable to attackers [47] [61] [63] and makes the existing vulnerabilities in hypervisors difficult to find and fix. Table. 2.1 illustrates the size of some of the main modern hypervisors [5] [17] [21] [37], some of which are comparable to a modern operating system. Some studies [6] [26] [50] have investigated the security of hypervisors and shown that many parts of the hypervisor are prone to be attacked.

However, embedded hypervisors ordinarily uses para-virtualization technologies, which can help reduce the influence of guest operating systems and is required to be designed with a small code-size due to the resource restrictions on

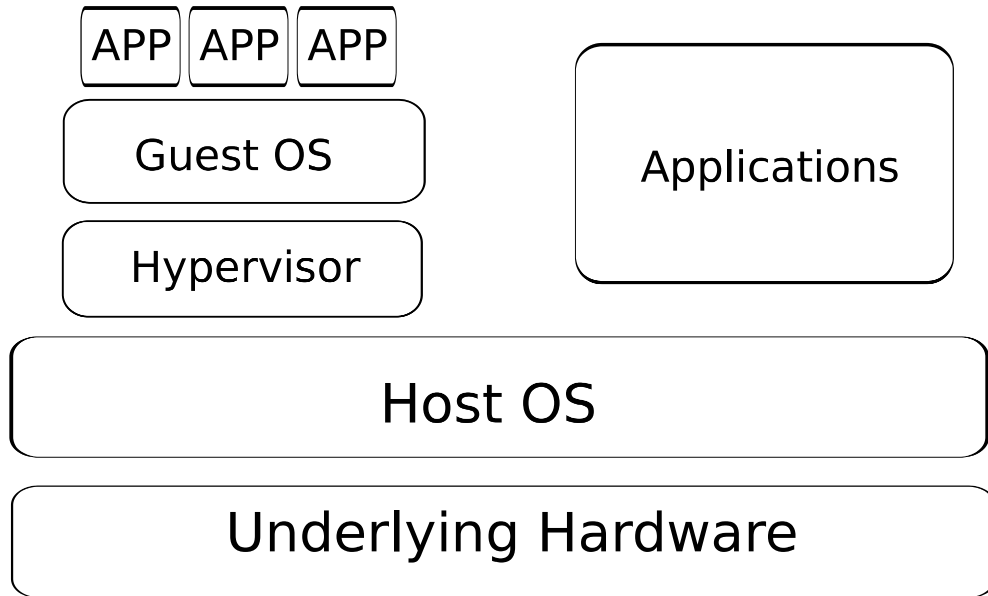

Figure 2.2: Hypervisors on a host operating system.

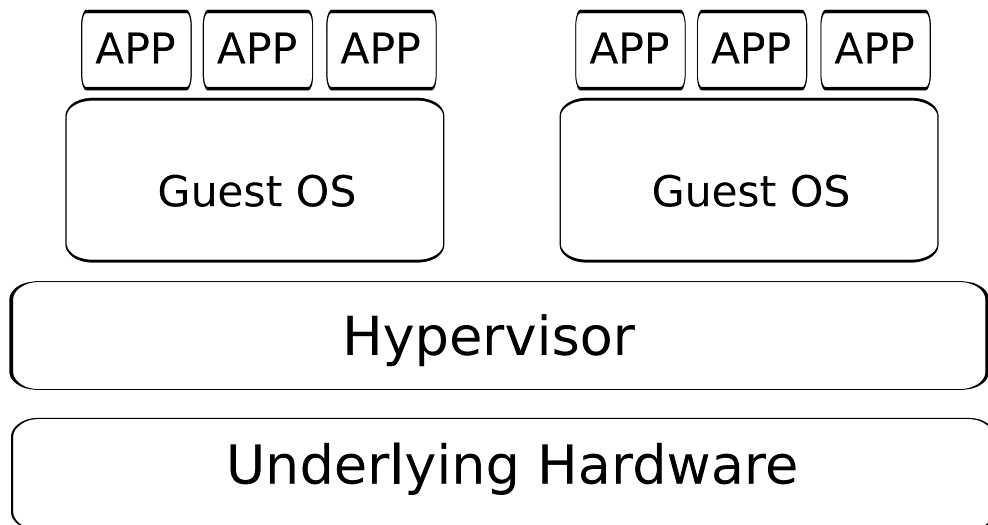

Figure 2.3: Hypervisors on rare hardware.

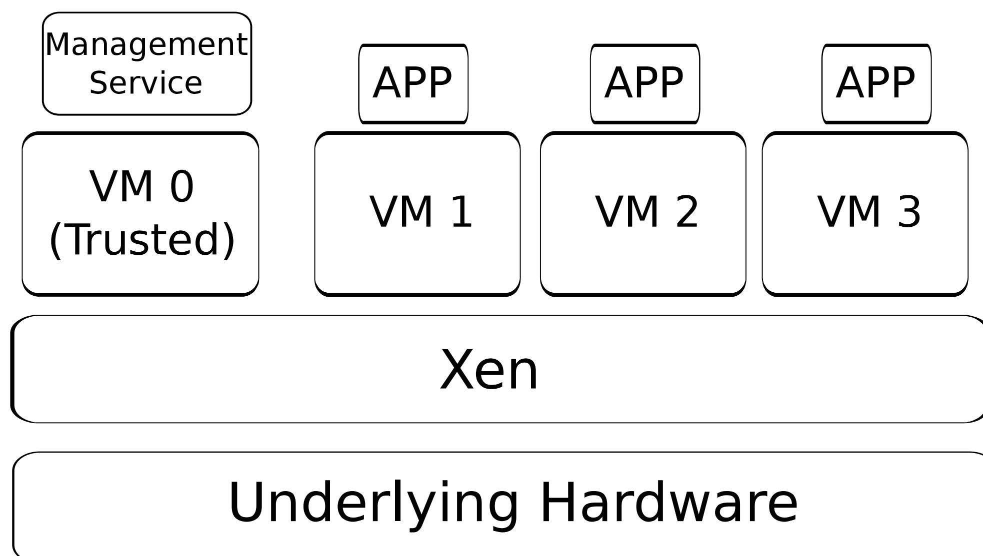

Figure 2.4: Xen hypervisor architecture.

embedded systems and the goal of security purpose [10]. However, embedded hypervisors sometimes do not manage all of the accesses of guest operating systems and trust behaviors of guest operating systems to avoid interrupting each other [48] [57]. These approaches assume that both the hypervisors and guest operating systems are trustworthy, which adds up to a large code size. Some embedded hypervisors [30] [36] [56] use the same method to control resource accesses from guest operating systems with traditional hypervisors, which would cause noticeable overhead for the performance of guest operating systems. More methods are still needed to optimize the performance of the embedded hypervisors, such as importing hardware virtualization extensions [69] or applying dedicated methods based on a specific processor architecture [43]. Microkernels, which may not be considered as a type of hypervisor, can also control hardware resource accesses from guest operating systems to isolate each guest operating system. However, microkernels usually require considerable modifications to the guest operating systems and may yield a heavy penalty to performance for the external pager. Embedded hypervisors also have similar vulnerabilities to traditional hypervisors, as discussed above.

2.2 Security Approaches based on Hypervisors and Peripheral Devices

Many researchers believe that reducing the size of the hypervisor can help to solve these issues and have proposed many methods to build a hypervisor with a small size of code that should be trustworthy.

BitVisor [61] focuses on the components of virtualizing I/O devices that enlarge the code size of hypervisors greatly and reduce the reliability of VMMs. It proposes a para-pass-through architecture to allow most of the I/O access to pass through the hypervisor and thus remove many functions for virtualizing I/O devices outside hypervisor to reduce the size. The guest operating systems running on the hypervisor can do directly access the I/O devices, with only the necessary operations monitored by the hypervisor. The evaluations are mainly performed using ATA storage devices, and the sizes of BitVisor and the para-pass-through driver for guest OSs are respectively only 20 KLOC and 1.4 KLOC, which are both significantly smaller than other hypervisors such as Xen (100 KLOC [17]).

NOVA [66] is designed and implemented as a thin and simple hypervisor to show that this approach can significantly narrow the vulnerable surface of the hypervisors and improve the overall security of the system. It can host multiple unmodified guest operating systems and consists of a single micro-hypervisor with a size of 9 KLOC that should be trustworthy and a VMM (20 KLOC). It uses fine-grained functional decomposition to split a hypervisor into a micro-hypervisor, root partition manager, multiple virtual machine monitors, device drivers and other system services, and it only grants the highest privilege to the trustworthy micro-hypervisor. This method can implement a hypervisor with a small code size to improve the security of hypervisors by narrowing the vulnerable surface.

Xoar [22] applies similar methods to another famous open-source hypervisor, Xen. Instead of splitting hypervisors, this approach tries to split the trusted VM (with a large code size), which has the root privilege to manage other VMs, into service VMs, each of which has its own simple purpose. The paper analyzes the architecture of Xen, splits it into small components and restricts the VM for

accessing the hypervisor. It is possible to reduce the code size of the trusted VM (a Linux, 7.6 million codes) to a much smaller size of 13.5 KLOC, upon an underlying hypervisor of 28 KLOC. A small size can reduce the potential vulnerabilities in the trusted VM to enhance the security of the whole hypervisor environment.

TrustVisor [39] is a special-purpose hypervisor that can provide code integrity, date integrity and secrecy for the assigned applications. It utilizes the features of modern processors from Intel and AMD to achieve both good performance and high security. It also uses the hardware component TPM to attest the secure boot of an application. Although it does focus on reducing the code size of the hypervisors, the base code size of it is still very small (approximately 6 KLOC), which makes verification feasible. The small size can also help to enhance the comprehensive security of TrustVisor.

From the above researches, we find that most methods for reducing the size of the trustworthy code involve splitting the hypervisors (or the trusted VM) into small components and only granting root privilege as needed. A small code size usually indicates a lower possibility of attackers finding bugs, and even after being attacked, fixing simple codes is much easier than finding vulnerabilities in millions of lines of codes in complex hypervisors. If we want to apply these technologies, the modern hypervisor absolutely needs to be redesigned. However, these solutions still have the issue that the trustworthy codes need to interact with malicious codes in the hypervisor to complete normal functions. These interacting operations may be exploited to attack hypervisors to break the root trust of the whole systems [7] [9]. This type of security threat is difficult to be eliminate in the current hypervisor architecture.

Formal verification such as Sel4 is another method for ensuring that a software layer is trustworthy. It uses formal specifications to define the correct behaviors of a operating system kernel and verify the kernel based on these specifications. If a hypervisor can meet all of the requirements from the specifications, we consider this hypervisor to be trustworthy for managing access to guest operating systems to provide isolation. However, this method requires a long development

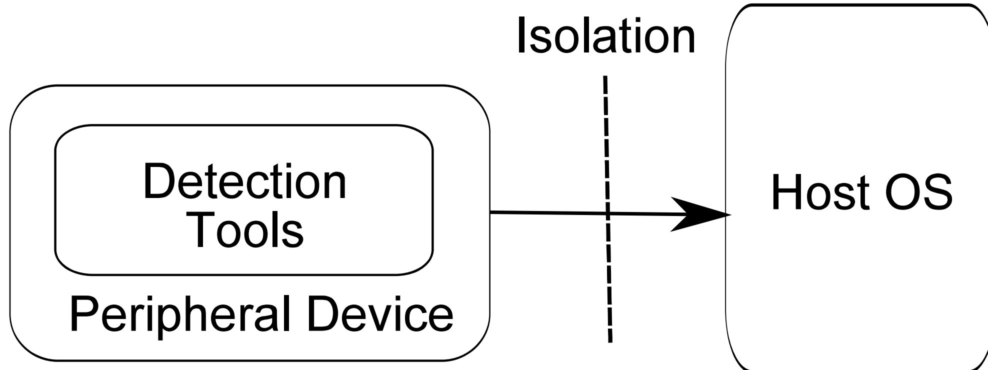

Figure 2.5: Detection based on peripheral devices.

period and substantial modifications to the existing hypervisors. Once the formal specifications are changed, the system needs to be reverified as trustworthy, which would also take extensive time. For modern hypervisors with a large code size, formal verification would require great design effort to change the current implementation of hypervisors or the trusted VM, which may be not acceptable for an enterprise hypervisor.

Some other approaches [15] [54] [77] are somewhat similar to hypervisors in building a secure environment based on hardware-centric methods (additional PCI cards or co-processors). Arati Baliga, Vinod Ganapathy and Liviu Iftode presented Gibraltar [15], a novel rootkit detection technique running on an external PCI card that automatically detects rootkits, which modify both control and non-control kernel data. It uses Myrinet PCI intelligent network cards to interconnect the Gibraltar and the target machine via a secure back-end network. The experimental results show that the Gibraltar can detect rootkits that modify both the control and non-control data structure while causing negligible performance overhead for the contention on the memory bus during detection tasks.

Copilot [54] is a co-processor-based kernel integrity monitor for the commodity systems proposed by Nick L. Petroni, Jr. et al. It is executed on a PCI add-in card that is connected to the host machine via an independent communication link to ensure the isolation. The copilot can detect malicious modifications to the host's kernel. The experiments illustrate that it can correctly detect certain real-world rootkits within a short time of their installation, with minimal penalty

to the host's performance.

These approaches are based on an additional PCI card to monitor the host OS, as shown in Fig. 2.5, and cause minimal overhead to the host OS, which meets the requirements of embedded systems. However, an additional computing device might be unsuitable for embedded platforms, which would add to the cost that is emphasized in embedded fields.

Kinebuchi et al. propose a machine architecture called Limited Local Memory (LLM) [34], which uses a hardware-centric isolation method that can efficiently protect of the detection tools and depends only on small-size codes to be trustworthy. The LLM architecture bases a small private memory area for respective core and provides a tamper-proof execution environment for a rootkit detector designed to monitor the target OS without becoming infected. Although the LLM architecture was not developed specifically for embedded systems, it provides some appropriate characteristics, such as low overhead, minimal modification to the guest operating systems and a small code size that should be trustworthy. We briefly introduce the LLM machine architecture to lead to our own research.

2.3 LLM Machine Architecture

The LLM machine architecture proposed by Kinebuchi et al. involves security enforcement on a multi-core processor. It requires a local memory region for respective core and an isolated Core0, which has the highest privilege to manage other cores and is designed to run a monitor OS in its self-only accessible local memory. Therefore, the monitor OS can execute an integrity checker to monitor the state of a target OS running on other cores without becoming infected. It uses a paging mechanism to execute the monitor OS in the local memory and emulates a local memory with 584 Kbytes, which should be large enough to accommodate the page tables for the monitor OS (256 Kbytes), the buffer data (128 Kbytes) for executing and the paging functions. It also proposes a bootstrap process to boot the system to a normal state. The architecture of the LLM is shown in Fig. 2.6, and the critical points in the LLM machine architecture are introduced as follows:

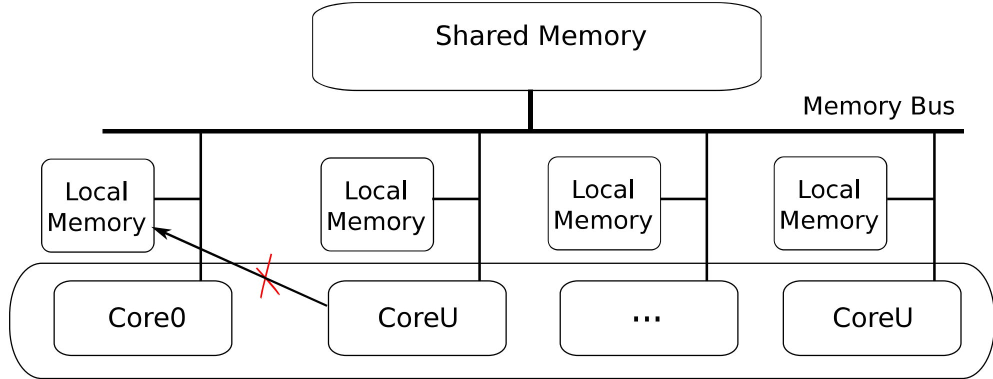

Figure 2.6: LLM machine architecture.

- Multi-core architecture processor. We need a special core (Core0) to run the monitor OS and there should be at least one more core (CoreU) for the target OS.
- Local memory for respective core. The local memory can be exploited to execute the monitor OS.
- Privilege function for isolating Core0. Core0 can be privileged to reset or halt other cores, and its local memory should not be accessed by the CoreUs. Therefore, the monitor OS running in Core0's local memory can avoid facing attacks from the CoreUs.

When the LLM machine architecture boots up, one core is selected as Core0 and is made privileged; it then executes the BIOS to load the bootloader into memory. The bootloader prepares the execution environment for the monitor OS and starts to run it. After running the monitor OS, Core0 begins to boot the target OS running on other cores. While booting up, it assumes that the BIOS and the bootloader can be trusted to work correctly to initialize the hardware and load the required data structure.

There are two critical points in the LLM machine architecture: local memory for executing the rootkit detector and core isolation to protect the rootkit detector. The two have somewhat similar applications in existing research.

The local memory approach is inspired by studies regarding the scratchpad memory and locked cache methods [16] [18] [55] [70]. Although caches can greatly

improve performance in desktop and server computing environments, cache invalidations usually introduce unpredictability into the performance of the application, which is very important in embedded systems, especially in hard real-time systems. The locked cache approach can help solve this problem by loading the assigned content of an application into a locked cache region to avoid cache invalidation during the runtime of the applications.

One study [70] combines both compile-time cache analysis and data cache locking to estimate the worst-case memory performance (WCMP) and provide an exact estimate of the WCMP of an application without heavy overhead. Another study [18] develops a generic algorithm to selectively load the content of an application to locked cache to obtain fully predictable performance with predictable overhead. Another paper, in turn [16], proposes the scratchpad memory method to provide a design alternative for locked caches, which can provide a performance and area reduction method for executing applications in embedded systems. A further study [55] compares scratchpad memories and locked caches in hard real-time systems and illustrates that the better choice depends on the architecture parameters (cache block size) and the application characteristics (basic block size).

The scratchpad memory and locked cache methods can selectively load the content of applications into scratchpad memory or a locked cache to reduce the energy consumption or improve the predictability of the application performance, which is significantly important in hard real-time embedded systems. These methods are similar to the local memory method, which executes applications in the on-chip memory. However, they do not address about the usage of on-chip memory for security purposes.

Some previous studies have applied core isolation on a multi-core processor as a security measure, such as NoHype [33], SecureCore [44] and the LLM machine architecture [34]. NoHype is designed to replace hypervisors for running multiple OSs on a multi-core platform. It assigns an isolated core and isolated hardware resources to each OS to provide complete isolation between OSs. It also assumes certain hardware features to avoid the denial of service attacks. However, because

of the strict isolation between guest OSs, this design cannot use one OS to check the state of another OS.

The SecureCore is somewhat similar in design to our research. It uses one secure core to run detection tools to monitor the other applications running on another core, while depending on a hypervisor to isolate the memory and other hardware resources. The detection tools can check the state of a control application running in an embedded system. SecureCore also describes some detection details to improve the accuracy of the detection tools for correcting the control application during the runtime. However, the isolation between cores still depends on a hypervisor, which may broaden the vulnerability of a system with extensive code.

The LLM machine architecture combines these two points to execute applications in an on-chip memory region (local memory) and protect the applications depending on the isolation of this memory region supported by the hardware architecture. The LLM machine architecture can ensure the security of the rootkit detector with minimal overhead and modification. However, the implementation of the LLM machine architecture is only emulated in a VMM QEMU and requires a limited but somewhat large amount of local memory (548 Kbytes) that may rarely be equipped on embedded platforms. Applying it to real embedded devices with a small-size on-chip memory would verify the efficiency of the LLM machine architecture.

Unfortunately, by now to our knowledge, there is not yet any hardware platform that perfectly meets all the requirements of the LLM machine architecture. Nonetheless, we have chosen a development board, RP1 [76], which is moderately similar to the LLM machine architecture. RP1 is equipped with a quad-core SH-4A processor and has a user memory region for each core that can be treated as the local memory, which meets the main requirements of the LLM machine architecture. However, it cannot provide a privileged core (Core0) with a self-only-accessible local memory, and other cores can also reboot the entire platform. We consider that this functionality can be equipped to a multi-core processor with modest effort during processor manufacture and production. Thus, we eliminate these requirements in RP1 and assume that the core used to execute the monitor

OS is privileged and isolated from the other cores. Based on the above assumption, we design an integrity checking system on RP1 to show the efficiency of the LLM machine architecture.

The size of the user memory equipped to each core in RP1 is 128 Kbytes, which is much smaller than in the QEMU emulation for the LLM machine architecture. The application of a paging mechanism to run an integrity checker within a fully functional OS (the monitor OS) in such a small memory region is a challenge in our research.

3

A Local-Memory-Based Extensible Secure Architecture for Embedded Systems

3.1 RP1 and SuperH Processor Architecture

We choose a development board RP1, [76], shown in Fig. 3.1 that has a SuperH-4A (SH-4A) processor with 4 cores each equipped with 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 includes 4 cores running at 600 MHz, each of which 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.

The SH-4A architecture has specific features that are useful for our design. Firstly, it manages pages faults using software and is equipped with a special memory architecture as shown in Fig. 3.2. The virtual memory space is divided into many areas, each of which has its own configuration for memory translation and caching. We can exploit this architecture flexibly to run different segments in separate areas to avoid or enable page faults. If we want to execute segments with

Figure 3.1: RP1 board.

page faults, we can simply jump to a start address in P0, while executing content in P1 without page faults. Similarly, the caching can be enabled or disabled as we choose, which is important in our design because we want to completely control page faults for the paging mechanism.

The software-managed TLB in the SH-4A architecture allows us to use any page table format in an OS that may be needed and to separate the TLB handler from the OS, which is very useful in our design when we execute an OS kernel in the direct address in P0. Not only do we know the virtual address that traps a page fault, but we also acquire the address that stores the page. For user processes that run from 0 at a virtual address, if we want to obtain the address of the page trapping the fault, we must add extra elements to the page tables to maintain the record. If we handle page faults in an OS externally, we must also acquire the necessary information about process execution in the OS.

The user memory equipped in each core can be accessed using assigned addresses by the underlying core, allowing us to selectively load segments of programs into for execution. With the isolation of this core and its local-memory, we consider the content running in the local memory to be reliable because other cores cannot touch this part for effective attacks. We next introduce the detailed

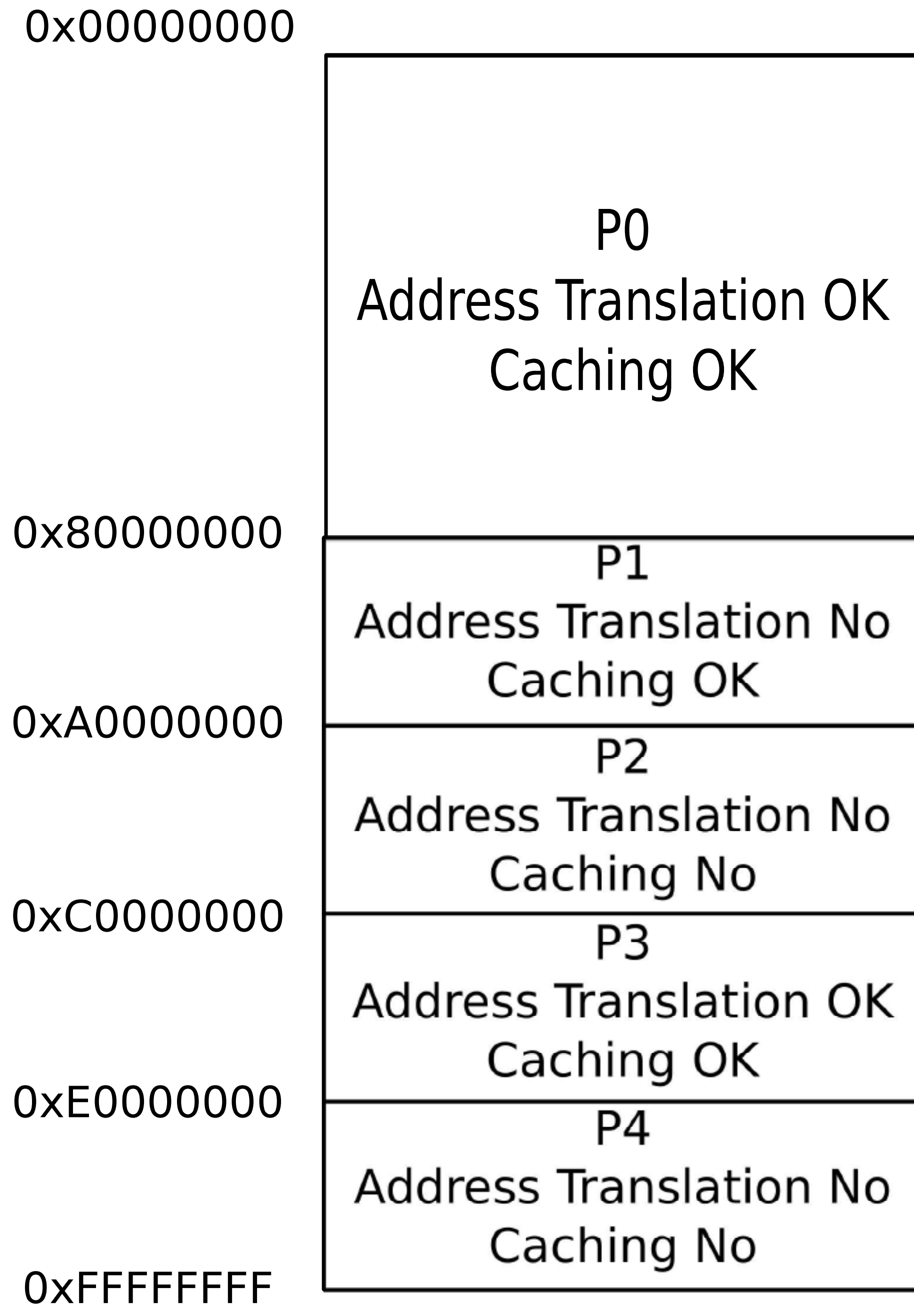

Figure 3.2: Memory architecture in SuperH.

method of our design.

3.2 Methods

In this section, we first propose the detailed design methods. Then, we analyze the threat model in our system. We focus on how to build a system to run a fully functional OS that executes an integrity checker in a small and isolated memory space. The checking technologies applied in the integrity checker and the vulnerabilities of the target OS are not the focus of the paper, and we thus discuss them only briefly discussion.

3.2.1 System Architecture

The basic system architecture applied to our research platform is shown in Fig. 3.3. We use a monitor OS to execute an integrity checker as a user process. The integrity checker can check the integrity of the shared memory used by the target OS. Importing an OS for the integrity checker offers two advantages: (1) the management of the integrity checker, such as executing, halting, sleeping or running parallel instances, is easy; and (2) other checking technologies can be ported to the monitor OS simply and effortlessly. The monitor OS running on Core0 executes the integrity checker to monitor the state of the target OS running on other CoreUs that provide applications for users.

The monitor OS (containing the integrity checker) runs in the local memory of Core0 and only uses the main memory to store swap content. A software component, the secure pager, is designed to provide a paging mechanism to swap in the needed content for running the monitor OS. Because the monitor OS and the target OS share the main memory, the swap content stored in the shared memory is vulnerable to attacks from the target OS. The secure pager also has the duty to verify the integrity of the swap content when it is swapped in. The secure pager is a critical component in our system, and we describe it in details below.

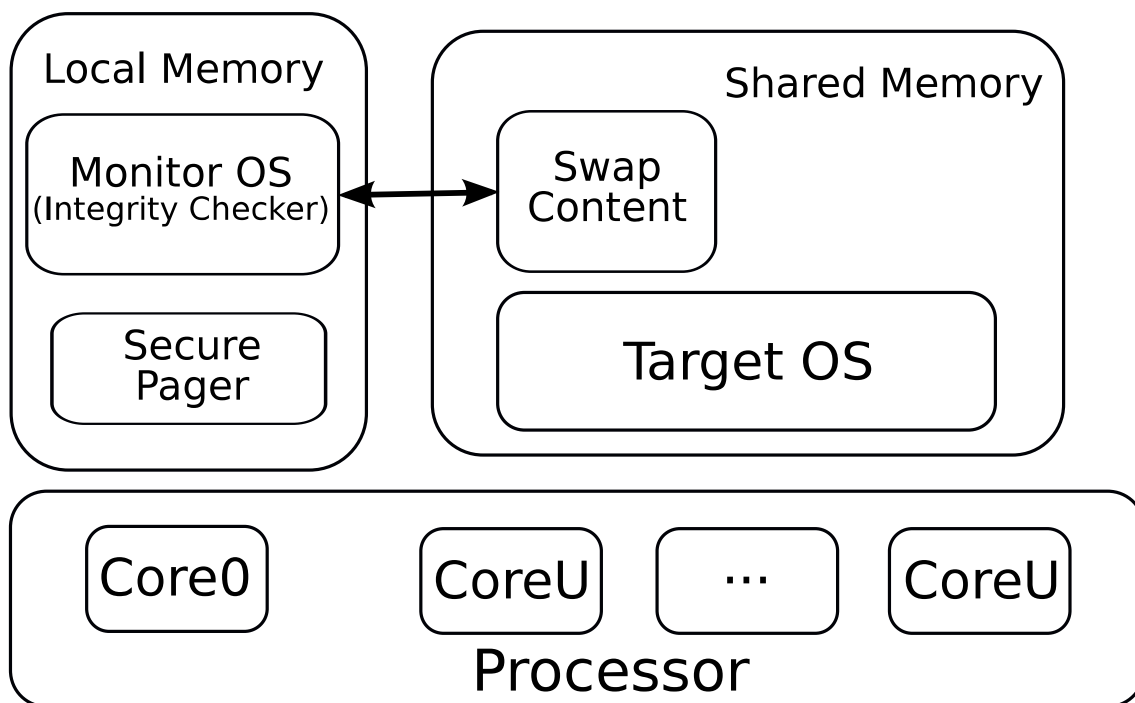

Figure 3.3: System architecture.

3.2.2 Secure Pager

To run the monitor OS in the local memory using a swap mechanism, the secure pager shown in Fig. 3.4 should take control of any page faults that occur in the monitor OS, judge whether a page has already located in the local memory or not and set the page tables of the monitor OS to the correct TLB mapping.

The secure pager located permanently in the local memory maintains a hash table for all the pages of the swap content in the shared memory, which is obtained before booting of the monitor OS to check the hash values of any content that must be loaded into the local memory to verify the integrity. If the integrity remains unchanged, the secure pager loads the content to run the monitor OS. Otherwise, the secure pager can detect the corruption, issue alarm messages about the compromise of the target OS and reboot the platform to restore the system to a normal state. We do not consider the resumption of the monitor OS during runtime in our prototype.

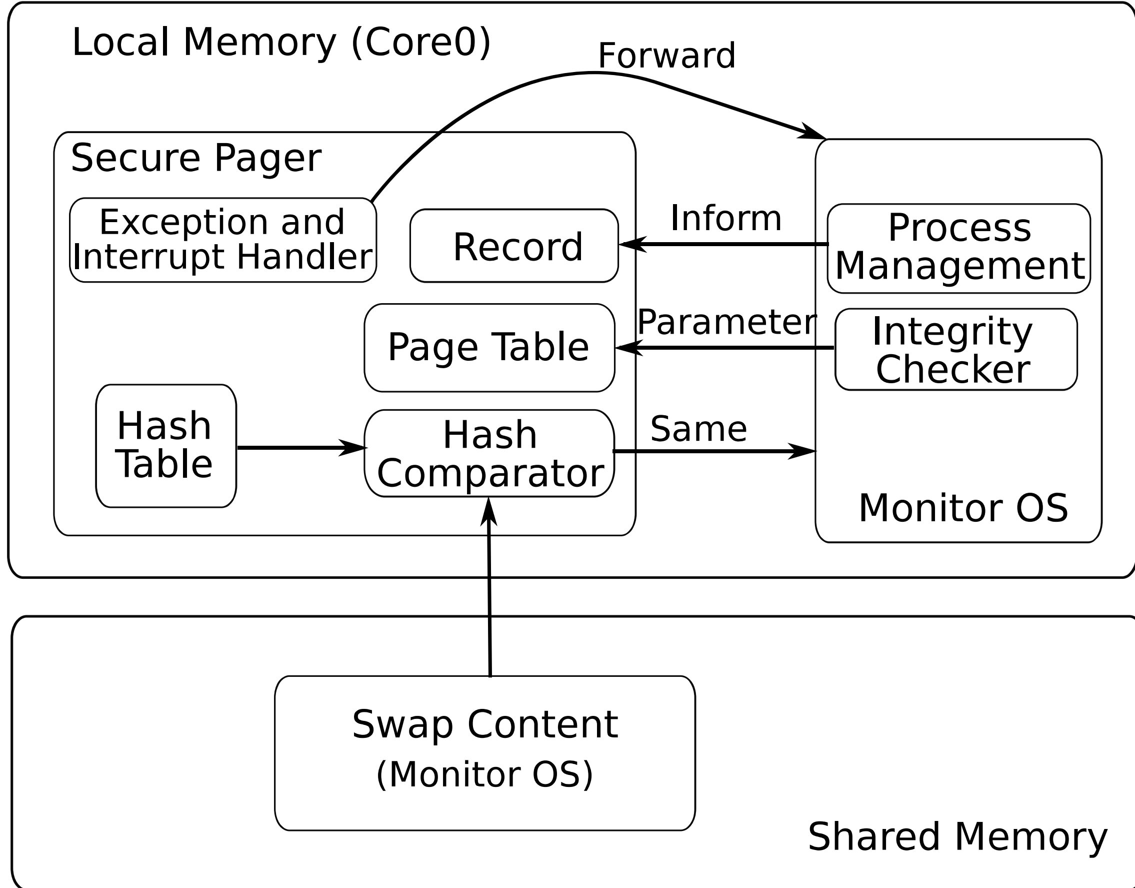

Figure 3.4: Work mechanism of secure pager.

In our system, we run the monitor OS at a virtual address that traps a page fault when the monitor OS first accesses a page. The secure pager handles the page fault, and judges whether this page belongs to the monitor OS kernel or to a user processor. It then loads this page into the local memory and checks its hash value. If the hash value is identical to the one computed in advance, the secure pager sets the page table according to the page type (kernel or user process) for the TLB mapping to run the monitor OS.

When the local memory is out of space and an extra page is needed for continued running, the secure pager searches the page tables of the monitor OS to choose a page to swap out. Because this page may have been changed during the running of the monitor OS, the secure pager calculates the hash of this page

again to update the hash table and swaps it to the original address in the shared memory that stores the page. It then clears this page and invalidates the entry in the page table and the TLB. Finally, the secure pager loads the needed page after it passes the integrity verification. To implement the swap-out mechanism, the secure pager should log the original address of any page loaded into the local memory. In our design, we deal with kernel pages and user process pages separately. We run the kernel of the monitor OS in the P0 address shown in Fig. 3.5, and therefore, page faults occur when the kernel begins to run. We can obtain the address in the shared memory that stores the kernel page from the virtual address, where the page runs. Conversely, the user processes run at a virtual address starting at 0x00000000. We can load the user processes directly into the local memory to execute them. However, this approach would require the secure pager to allocate some parts of the shared memory to swap out user processes if the local memory runs out of space. We use another method to allocate portions of the shared memory space for the user processes. We allow the monitor OS to manage a small region of the shared memory and first load the user processes into this part. We then use this region for the paging mechanism shown in Fig. 3.5. The paging mechanism requires the responding relationship between the virtual address where a user process runs and the physical address that stores the user process. We use page tables to log this correspondence relationship.

We design page tables for the monitor OS in a one-level page table format. Because of the software-managed TLB in RP1, we can design page tables freely to reduce the size or improve the performance. In our work, we use two types of page tables for the monitor OS shown in Fig. 3.6: the kernel page table and the user process page table. One entry in the kernel page table contains a mapping between a physical address (page address in the local memory) and a virtual address (equal to the original address in the shared memory that stores the page). One entry in the user process page table adds an extra factor called the intermediate address and thus consists of three parts, a physical address (page address in the local memory), a virtual address (page address in user memory space) and an intermediate address (original address in the shared memory). These page tables help the secure pager complete both the swap mechanism and the TLB setting.

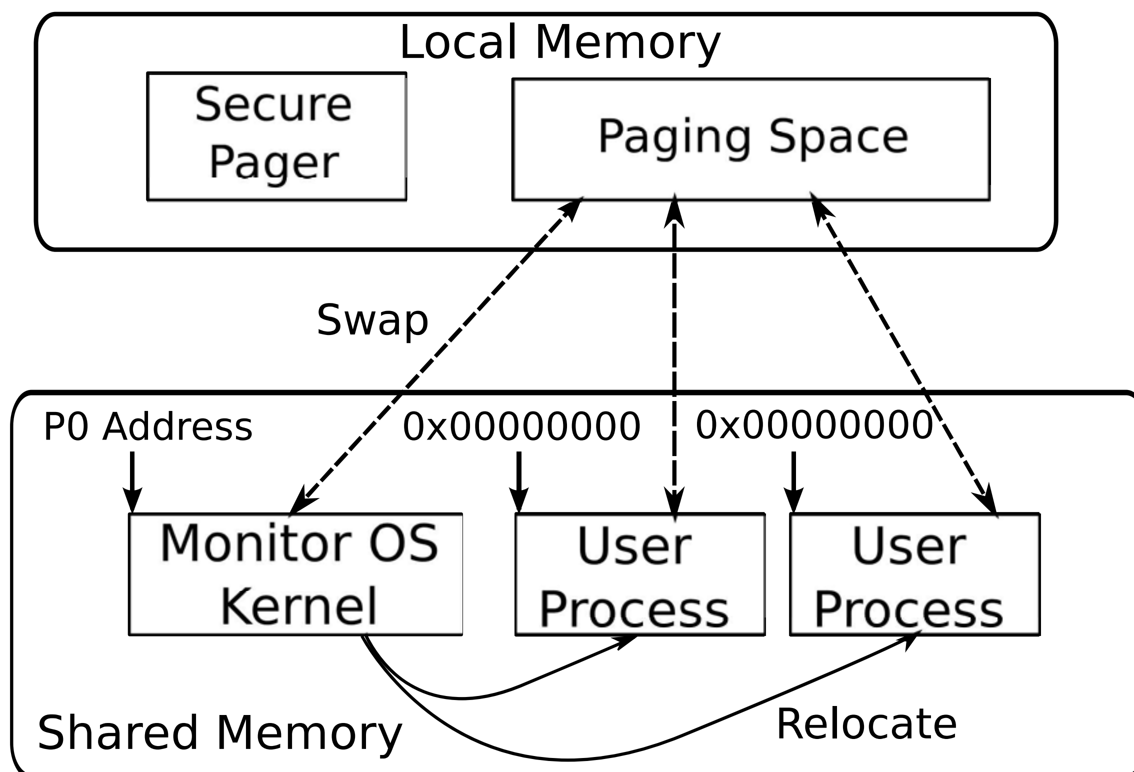

Figure 3.5: Execution address for the monitor OS.

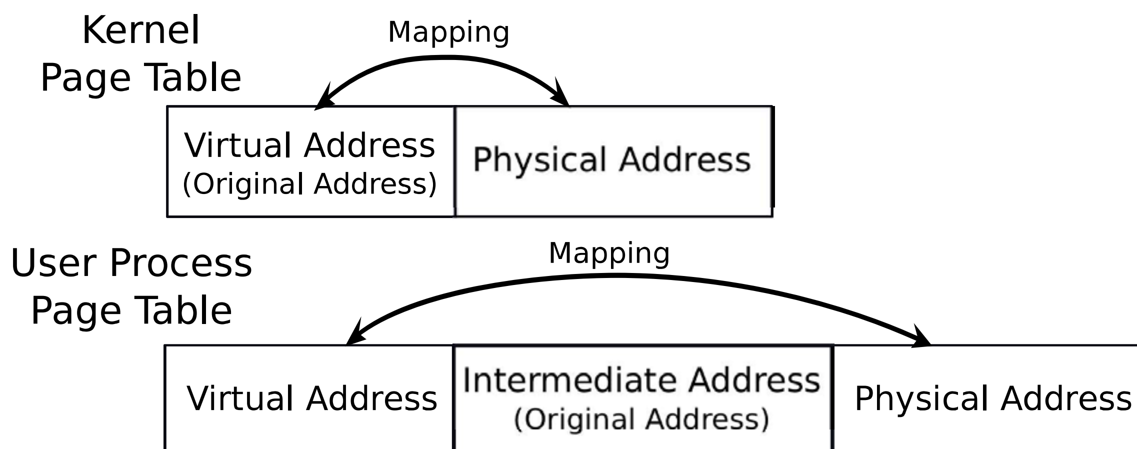

Figure 3.6: Page table format of the monitor OS.

The secure pager must activate one of multiple user process page tables when the monitor OS switches to another user process or executes a new one. To choose the correct page table for the corresponding user process, the secure pager

traps the operations of the user processes in the monitor OS and maintains a record. Therefore, when it needs to activate the page table for use, the secure pager determines that which table has been prepared for the current user process. With this trapping, the secure pager can choose the corresponding page table to execute the current user process in the monitor OS.

3.2.3 Reduce Size of Page Tables of the Monitor OS

There is a single kernel page table and may be several user process page tables in the secure pager. We can restrict the number of user processes that can run concurrently in the monitor OS to control the number of user process page tables, which can reduce the space in the local memory that the page tables occupy. Furthermore, the special format used in our work can further reduce the size of the page tables. In some cases, a single user process may be large enough to require larger page tables. We prefer to split this type of user process into smaller processes and execute them sequentially to keep page tables within a small size. With these methods, we can apply our system within even a very small local memory. We then compare our method and the typical two-level page table in the x86 architecture.

The x86 architecture with hardware-handled TLB ordinarily uses the type of two-level page table shown in Fig. 3.7, which contains a page directory (PD) to store addresses that point to page tables (PTs) and PTs that store physical addresses (PA) to transfer a virtual address to a physical address, and the format of the page tables is fixed. If a user process with 8 Kbytes runs in the x86 architecture with a page size of 4 Kbytes, it first allocates a page with 4 Kbytes for the PD, and then allocates another page as a PT to store 2 entries for the user process. The size of the page tables for this process adds up to 8 Kbytes. Conversely, in our method, using a page size of 4 Kbytes, a one-level page table means that each page of a user process only needs a page table entry with 8 bytes, and we add another element to every entry in the page table to store the swap-out address, which occupies another 4 bytes for each entry. Therefore, a user process with 8 Kbytes occupies 24 bytes for its page table. This size difference decreases

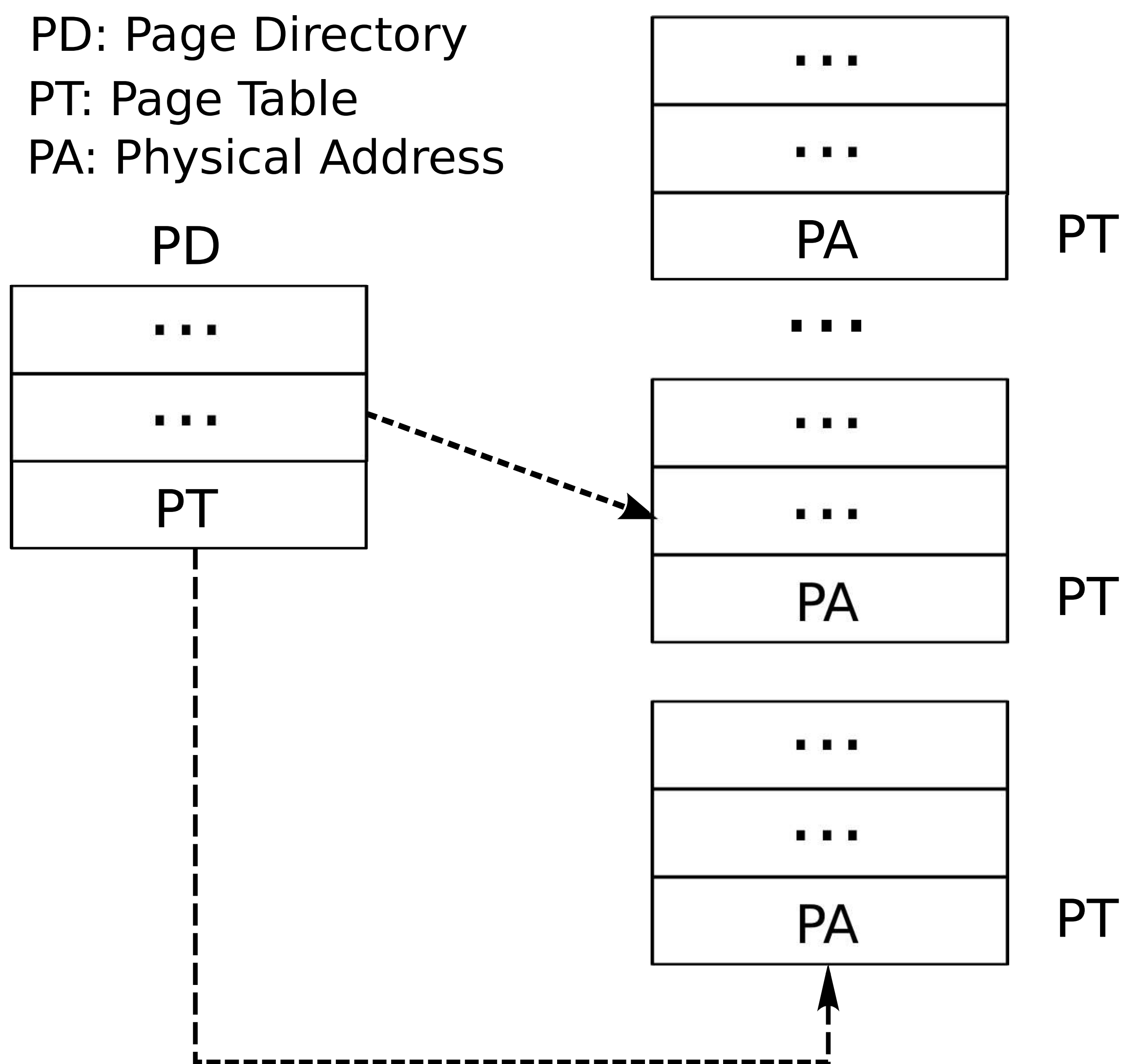

Figure 3.7: Two-level page table in x86 architecture.

when the size of the user process increases, as shown in Table. 3.1. However, we prefer to use smaller user processes or split large user processes into small pieces so that the page tables occupy a small space in the local memory. With the restriction on the number of user processes that can concurrently run in the monitor OS, we can efficiently control the size of the page tables to ensure that the secure pager has a permanent location in the local memory for executing the monitor OS.

The secure pager should also have certain other functionalities, such as mem-

Table 3.1: Page table size comparison (Bytes).

User Process Size	8K	16K	512K	1M
x86	8K	8K	8K	8K
Our Method	24	48	1.536K	3.072K

ory management of the local memory, hash computation, hash table maintenance and exception and interrupt handling, to assist the paging mechanism. An internal scheme is shown in Fig. 3.8 that further elucidates out secure pager design.

3.2.4 Extensibility

When we need to fix bugs, make optimizations or add new features to the integrity checker, the normal running of the target OS should not be disturbed. Automatic updating is a good method to solve this problem. The main purpose of the automatic update is the security. The update procedure must be safe enough to ensure that the integrity checker is updated correctly. We add an automatic update function to the secure pager, which is trusted for permanent existence in the local memory. The details of this method are discussed in the section on implementation. Using this function, we can freely extend the functions of the integrity checker to perform optimizations or add features.

3.3 Threat Model

3.3.1 Assumption

Firstly, we want to propose some assumptions about the threat model, including the following:

- The internal hardware configuration is trustworthy, including isolation for Core0 from the CoreUs and that Core0's local memory is accessible only to itself.

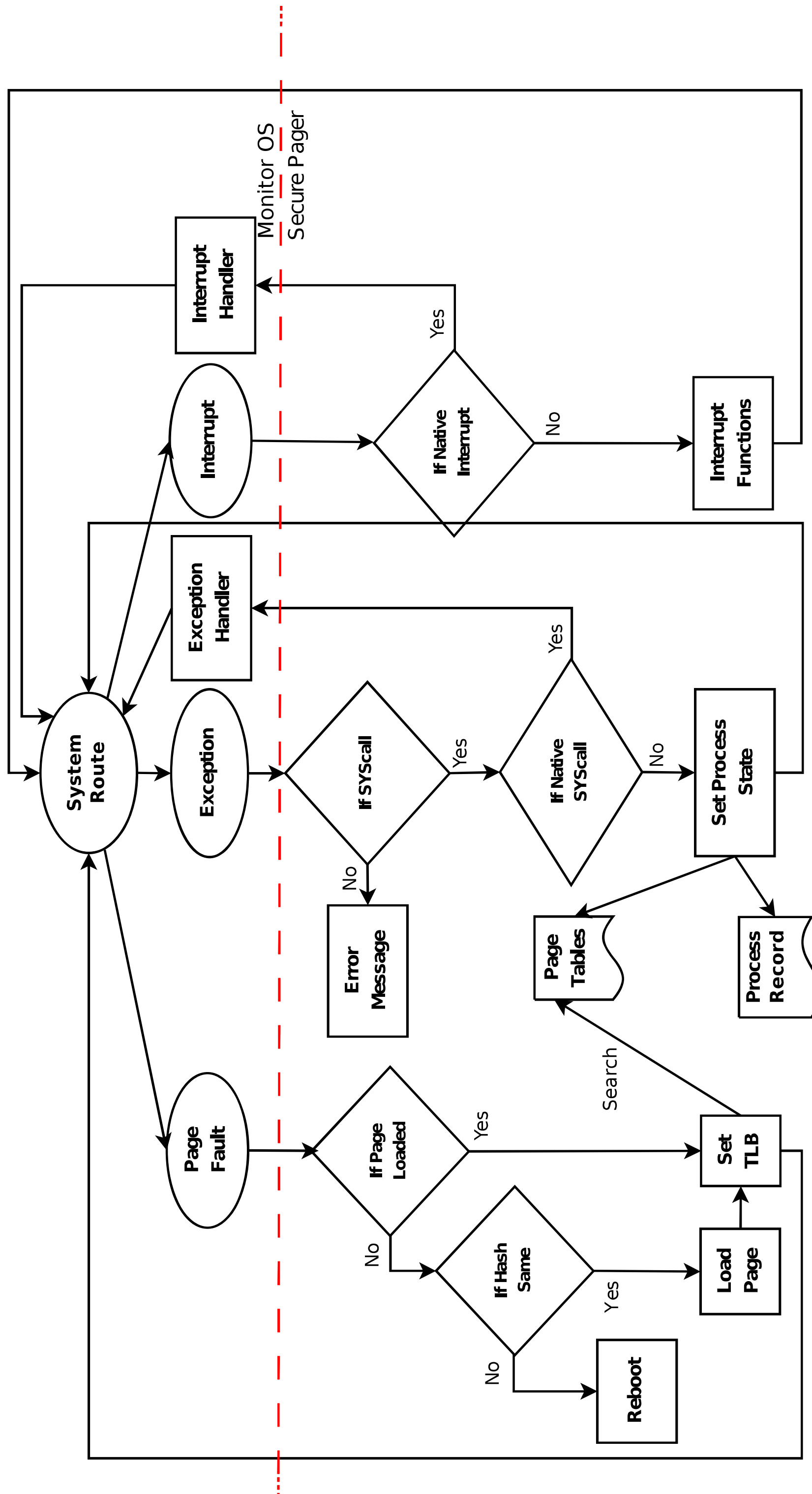

Figure 3.8: Internal scheme of secure pager.

- Because the RP1 loads a system image from a remote server via the network in our research, the remote server is managed with good security, and the network connection and hardware firmware are trustworthy. Therefore the firmware can receive a correct system image and boot to a normal running state.
- The secure pager and the monitor OS are designed well to avoid not malice.

We do not assume other hardware features in our architecture that may help to construct our architecture or improve the performance. For example, if the hardware can provide read, write and execution protection on a memory area for assigned cores, we can assign that CoreUs cannot read, write and execute any content in the swap content region. In this way, we do not need to concern the security of the swap content and this will improve the performance because there only needs a pure paging mechanism without any other protection methods. In our research, we do not concern these features and try to construct our system with the basic hardware assumptions.

Based on these basic assumptions, we can ensure that our system can boot to the normal state and eliminate hardware attacks and vulnerabilities that exist during the image loading and network connection. Our discussion focuses on the runtime security of our architecture. We divide the security of this architecture into three parts: the security of the target OS, the security of the monitor OS and the security of the secure pager.

3.3.2 Security of the Target OS

In our system, the vulnerable component is the target OS, which faces many attacks in a real computing environment and may be infected. There are already many form of malware that can attack OSs to control them or steal sensitive information. Rootkits are a popular method for attackers and are easy to obtain. Our method can easily be used for protection against these rootkits if we know their mechanisms; however, we cannot foresee future rootkits or other malware technologies and the rootkit checker may become obsolete. Therefore, there is

a tendency to adapt a generic method for protecting the target OS instead of specific rules to protect against individual rootkits. The generic methods monitor the target OS in a secure environment over a long period and obtain necessary information representing the integrity of the target OS during runtime. This information can also be used in our architecture to ensure the correct execution of the target OS, which is a more broadly applicable method for monitoring different OSs in various platforms. It is important to recall that it takes a long time to obtain enough data to represent the integrity of a running OS. Furthermore, when more data are required, the monitoring time is increased substantially, which adds to the time-cost of development. It may be more efficient to combine rootkit detection for existing rootkits and integrity checking for future unknown vulnerabilities to maintain a reasonable development time. In our work, we refer to both of these methods as integrity checking within an integrity checker.

The monitor OS checks the shared memory content used by the target OS to perform the integrity checking. Thus a concrete rootkit detector or the generic method to check the integrity can be applied in our architecture. We consider that checking technologies based on shared memory inspection are suitable for the integrity checker to monitor the target OS. However, in our architecture, we do not provide a secure hardware mechanism for Core0 to obtain the register information of other CoreUs, and this is a natural limitation for the integrity checker. If the malware infects the system via the internal registers, the integrity checker is restricted to detecting this kind of attacks in our architecture. We can modify the target OS to dump the internal registers into the shared memory during runtime, but we cannot make sure that the infected target OS can complete this task correctly. Therefore, it is difficult to provide an effective method for Core0 to obtain the correct information about other cores' registers. However, an integrity checker based on hypervisors can acquire this information because hypervisors can access the underlying hardware, including each core in the processor.

Although the security of this part is related to users, it is not the most critical part in our architecture. Even if the target OS is infected, the integrity checker is still monitoring it and may detect that the target OS is no longer trustful. The integrity checker would do some related operations based on the functionalities

equipped on it. It should firstly try to recover the target OS to the normal state and then if it cannot resume the target OS from a fatal or unknown security vulnerability, it would reboot the target OS to avoid this kind of attacks.

3.3.3 Security of the Secure Pager

Because of the secure pager is located permanently in the local memory, its security depends on the hardware architecture in the RP1. We assume that one core in the RP1 is selected as Core0 and can be isolated well from the other cores with self-only-accessible local memory. As long as the local memory cannot be accessed by the other cores, the secure pager can implement a completely secure paging mechanism to run the monitor OS.

Although the security of this part is most important in our architecture, because the hardware assumptions can assure the security of this part, we do not need to concern about it in our research.

3.3.4 Security of the Monitor OS

During the normal running, the target OS should never access the shared memory space used to store the swap content of the monitor OS. However, the target OS can access this part, and may be infected with instructions to read or modify the swap content. Read attacks are less critical in our architecture because they reveal the messages about the monitor OS but do not influence its normal running. The write attacks are much worse than read attacks in our system that may disturb the running of the monitor OS. The monitor OS would execute infected pages for the detection tools. However, both of these attacks are dangerous for the security of the monitor OS.

The secure pager is used to detect these attacks. The encryption on the swap content in the shared memory is exploited in the secure pager, and therefore, even if these pages are read, the target OS cannot acquire any meaningful information. However, the secure pager can check the integrity of the swap content to be loaded into the local memory. If write attacks occur, the secure pager can detect them

and reboot the system to prevent the infected monitor OS from continuing to run. Our method can ensure that only the correct monitor OS (the integrity checker) can run in the Core0's local memory and the integrity checker can monitor the state of the target system.

The security of this part is also very important in our architecture. Although the read attacks cannot disturb the normal running of the integrity checker, once the write attacks occur, the integrity checker may be unable to be executed correctly in our system. This violates the basic feature of our secure architecture, and the integrity checker should only be running in a normal state. We can try to place multiple backup images in the shared memory to restore the modified content. However, it is difficult to make sure that these backup images are not attacked by the target OS. Under this situation, we cannot 100% recover the integrity checker. We prefer to reboot the whole system to ensure the correct execution of the integrity checker, which is the basic feature of our secure architecture to improve the overall security.

After the system image is loaded into the shared memory for booting, the bootloader inside of it loads the secure pager into Core0's local memory and loads the monitor OS and the target OS into the shared memory. It then transfers control to Core0 to boot up the secure pager to execute the monitor OS. After completing the boot process of the monitor OS, Core0 informs CoreUs to begin running the target OS. During runtime, the monitor OS runs the integrity checker to monitor the state of the target OS and the target OS runs various applications to meet user requirements. If the integrity checker finds that the target OS is infected, it handles this vulnerability depending on the functionalities equipped to it. If the infected target OS has modified the swap content of the monitor OS, we consider the root trust of our system to be compromised. The secure pager would reboot the entire platform to avoid this type of attack.

3.4 Implementation

In this section, we first introduce the implementation environment of our system. The integrity checking, execution and automatic update of the integrity

checker are then illustrated. Finally, the system architecture using two schemes is proposed.

3.4.1 Implementation Environment

Because we focus on applying our architecture to embedded systems, possibly including real-time systems, it is better to use a passive integrity checker without disturbing the normal running of the target OS to reduce the overhead. We choose an integrity checker that can check the entries of the Linux system call table and detect the *hide_task* rootkit that uses the DKOM (Direct Kernel Object Manipulation) mechanism to infect the kernel data [59].

We select a Unix-like OS called xv6[1] as the monitor OS. Xv6, developed at M.I.T., is very simple and provide a virtual memory functionality. It was originally written for the x86 architecture, so we ported it to an SH-4A architecture for our research. To port xv6, several modifications were required.

- The boot loader should be removed because, unlike the x86 architecture, the RP1 can load a kernel directly to a fixed address to boot. There is no BIOS in the SH-4A architecture. The firmware will perform the primary initialization of the RP1 and load the system image to the shared memory, and the RP1 will execute this image to boot our system.
- The processor initiation and context switch must be modified. The assembly used for the processor initialization and the context switch is architecture specific, and they must be reimplemented using the SuperH assembly to reimplement these parts.
- Exception and interrupt handlers need to be added at specific addresses. The exception and interrupt handlers in SH-4A are assigned to locations at specific fixed addresses; therefore, they must be moved to these addresses. These parts must also be rewritten with a specific assembly to run on the SH-4A architecture

[1] Xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2012/xv6.html.

- Because RP1 does not support IDE devices, the IDE driver is removed from the kernel, and the file system is modified to use other types of devices. We select a RAM file system (*fs.img*) that is attached to the kernel to solve this problem. The output function *cprintf()* needs to be modified to use the *SCIF* port, a serial controller, to print debug messages.

- Xv6 is intended to run on one core of the processor. Removing the multi-core support makes the code simpler and easier to control.

After porting, we divide xv6 into two parts: the kernel and the RAM file system. We prefer not to use the same storage as the target OS to narrow the vulnerable surface. xv6 uses 1 Mbytes of the shared memory space to store the kernel and the RAM file system, and another 1 Mbytes space to store the swap content of the user processes.

We choose an embedded Linux with kernel version 2.6.16 as the target OS because embedded Linux supports the SH-4A processor architecture. The embedded Linux uses 64 Mbytes memory space in the RP1 board and a network file system exported by a remote server. The embedded Linux requires minimal modifications to avoid using Core0 or touching the shared memory region that stores the swap content of xv6.

We write a secure pager from scratch. It includes many functionalities such as a paging mechanism, SHA1 hash computation, page table maintenance, page fault handling, TLB setting, local memory management and other assisting functions. The prototype implementation occupies less than 44 Kbytes in the local memory, which contains a hash table of 10 Kbytes and the page tables for xv6. The size of the secure pager may vary, as we may add more functions in the future, or the configuration of the number of user process page tables may change. However, we can make further optimizations and restrict the number and the size of concurrently running user processes in xv6 to ensure that the size of the secure pager remain acceptable for permanent location in the local memory. We next provide a more detailed description of the code for the secure pager.

The secure pager contains many files with different functions. The number of lines of code (LOC) of each file are shown in Table. 3.2. The code can be divided

Table 3.2: LOC of files in the secure pager.

File	LOC	Function
kernel.S (assemble)	516	Exceptions and interrupt routing
exp.c	521	Exception handler (mainly TLB handler)
kalloc.c	87	Local memory space management
main.c	33	Boot the secure pager
print.c	213	Print out debug messages
proc.c	330	Log operations of user processes
put.c, scif.c	52	Scif interface driver
string.c	125	Memory copy function
dtu.c	37	DTU driver
md5.c	242	MD5 hash mechanism
sha1.c	228	SHA1 hash mechanism
RSA.c	831	RSA algorithm
RC4.c	200	RC4 algorithm
.h	1270	Head files for programs
Makefile	40	Make for the system image
Total	4804	

into many types. The critical functions, including exception and interrupt system routing (516 LOC), the TLB and exception handler (521 LOC), the encryption and decryption mechanism (RSA and RC4,1031 LOC), hash function (SHA1 or MD5, 242 or 228 LOC) and debug function add up to 4804 LOC. For comparison with Kinebuchi's research, we show the LOC from his research in Table. 3.3. His paper does not give many details regarding the functions in his LLM architecture; therefore, we can only compare the total code and the similar functionalities. The total LOCs are on similar levels and a similar mechanism can be implemented using an approximative number of LOC. We add more functionalities to enhance the security of our system and extend the monitor functions during runtime, but these codes can still be kept to a small size. To limit the size of the local memory, we do not add too many functions to our secure pager, which may need to be formally verified to be granted higher security. These codes are sufficient to meet the requirements of our architecture.

The host machine is running Ubuntu 13.04 with a quad-core I2.66 GHz ntel Core 2 processor Q9400, 4 GB RAM and a 320 GB hard disk. We use the host

Table 3.3: Code size in Kinebuchi's research.

Types	LOC
Change to xv6	404
Secure pager	592
SHA-1	539
OS loader	146

machine to compile the system boot image and complete the encryption part in the automatic update.

3.4.1.1 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 integrity checker. 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 [71]. 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.

3.4.2 Integrity Verification of Integrity Checker

In this section, we propose how the integrity of the integrity checker is verified. When the integrity checker 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 verification in our system.

We apply MD5 and SHA1 hash algorithms in the system. MD5 is not so robust in [73]. 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. 3.9 is introduced as following:

- Firstly, hash values of all the pages of the integrity checker are calculated by the secure pager and stored into the hash table that exists permanently in the local memory. Then, the integrity checker is encrypted, and the secure pager starts to exec1ute it.

- When a page of the integrity checker 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 integrity checker 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 integrity checker.

- When all the space in the local memory is used and there needs to load in a page for the integrity checker, 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 integrity checker 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

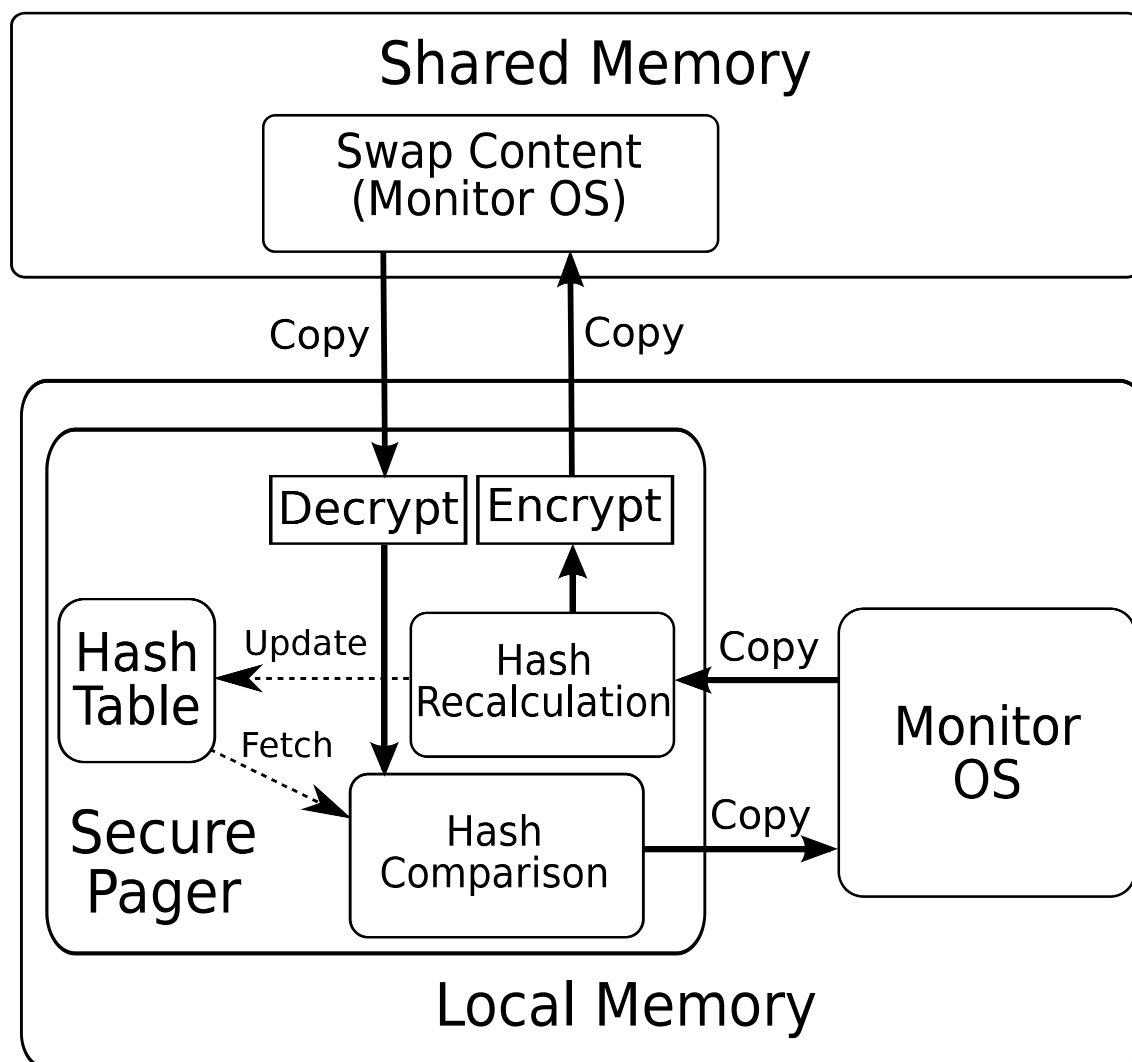

Figure 3.9: Integrity checking.

shared memory and free this page. Then, the needed page can be loaded into the local memory as the previous step.

3.4.3 Execution of Integrity Checker

The integrity checker 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 integrity checker, such as sleeping, exiting, or running

multiple monitoring instances.

To solve this problem, we use xv6 [3] as the monitor OS to execute the integrity checker as a user process. Xv6 can provide some standard APIs and virtual memory functionality. With xv6, we can develop or optimize an integrity checker 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 larger) that contains the integrity checker and other files.

3.4.4 Automatic Update of Integrity Checker

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 as the target OS for loading the update file to shared memory, from where the secure pager can acquire it, thus 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 integrity checker in a secure manner.

A safe update procedure shown in Fig. 3.10 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

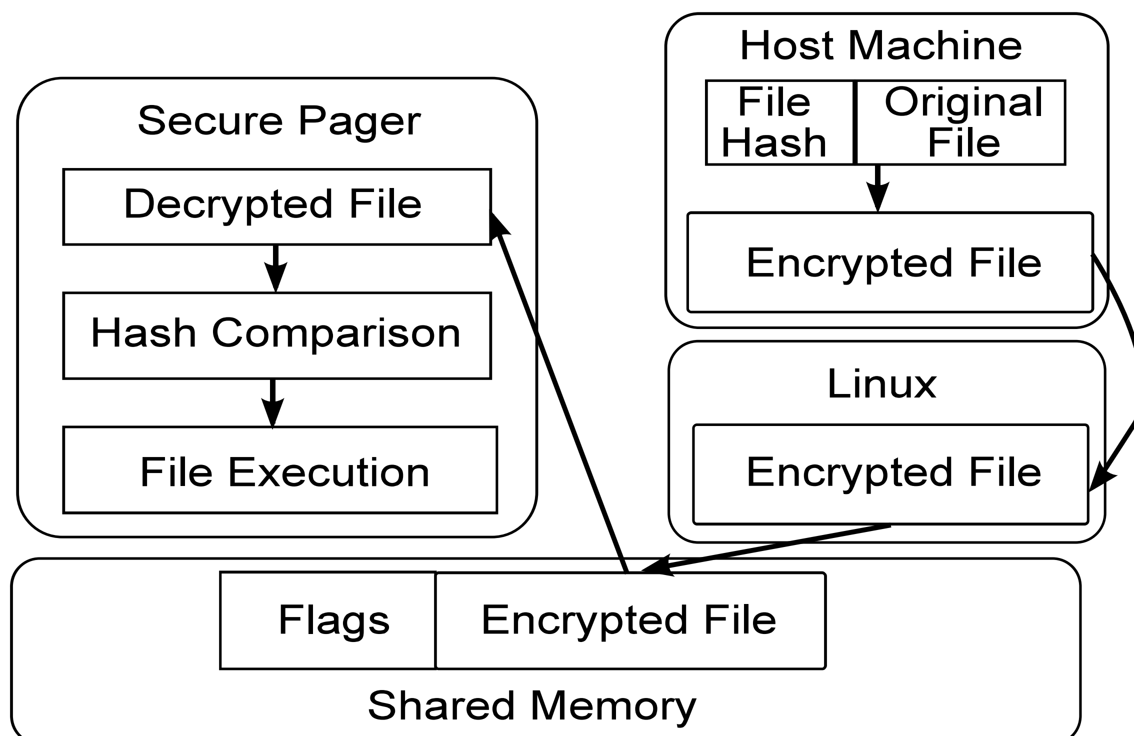

Figure 3.10: Automatic update.

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 integrity checker. 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 large integrity checkers, even larger than the local memory.

During the automatic update process, how the decryption key is stored is the key problem to keep the 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. 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 should remind that this procedure can only ensure that only the correct update file can be applied in the monitor OS. We do not equip the mechanism to ensure that this update is successful. We consider that the system manager or maintainer should take this responsibility to confirm that the system has been running the update file. If the update procedure failed with the cracked update file, the secure pager can alarm this fail but cannot obtain any information about the update file or redo the update. The system manager or maintainer should try to restart the update procedure and make sure that the latest integrity checker is running in our system.

We can adopt two ways to update the integrity checker: single integrity checker pattern (S-pattern) and multiple integrity checkers pattern (M-pattern). The S-pattern only runs one integrity checker in the system. Therefore, the space in the local memory used by the integrity checker is small, which is valuable due to the size of the local memory. However, the update introduces an interruption between ending the old integrity checker and starting the new one. The halt of the integrity checker may be a security hole in our system. On the other hand, the M-pattern executes multiple integrity checkers 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 integrity checker without disturbing the monitoring functions. However, we have to replace the old integrity checker 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 integrity checker 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 integrity checker in our system. We will introduce system schemes in the following section.

3.4.5 System Architecture

In our research, we propose two main system schemes. Firstly, we implement the system architecture with an nature-straight scheme. However, we intend to find some manners to improve the performance of the original scheme. We call the modified one the optimized scheme. The details of these two schemes will be introduced in the following part.

3.4.5.1 Original Scheme

The original scheme is shown in Fig. 3.11. 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 integrity checker. After the integrity checker starts, Linux will start to provide applications for users. The integrity checker 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 integrity checker. If the integrity checker is compromised, the secure pager can detect it and reboot to restore the whole system. With the integrity verification and the automatic update of the integrity checker, it can provide a reliable system with extensibility.

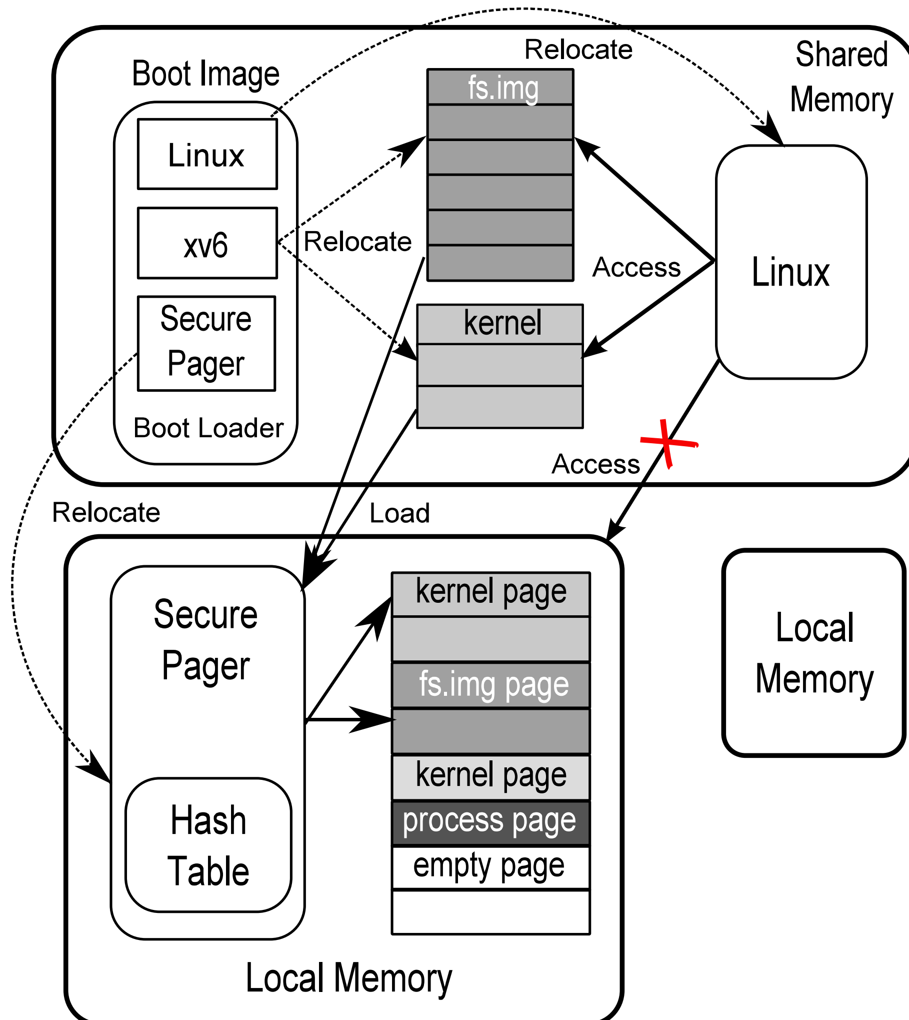

Figure 3.11: Original scheme.

3.4.5.2 Optimized Scheme

Based on the original scheme, we propose an optimized scheme with changes of the xv6 relocation shown in Fig. 3.12. As mentioned above, we know that xv6 is divided into a kernel part (small) and a RAM file system part (*fs.img*, comparable large). 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

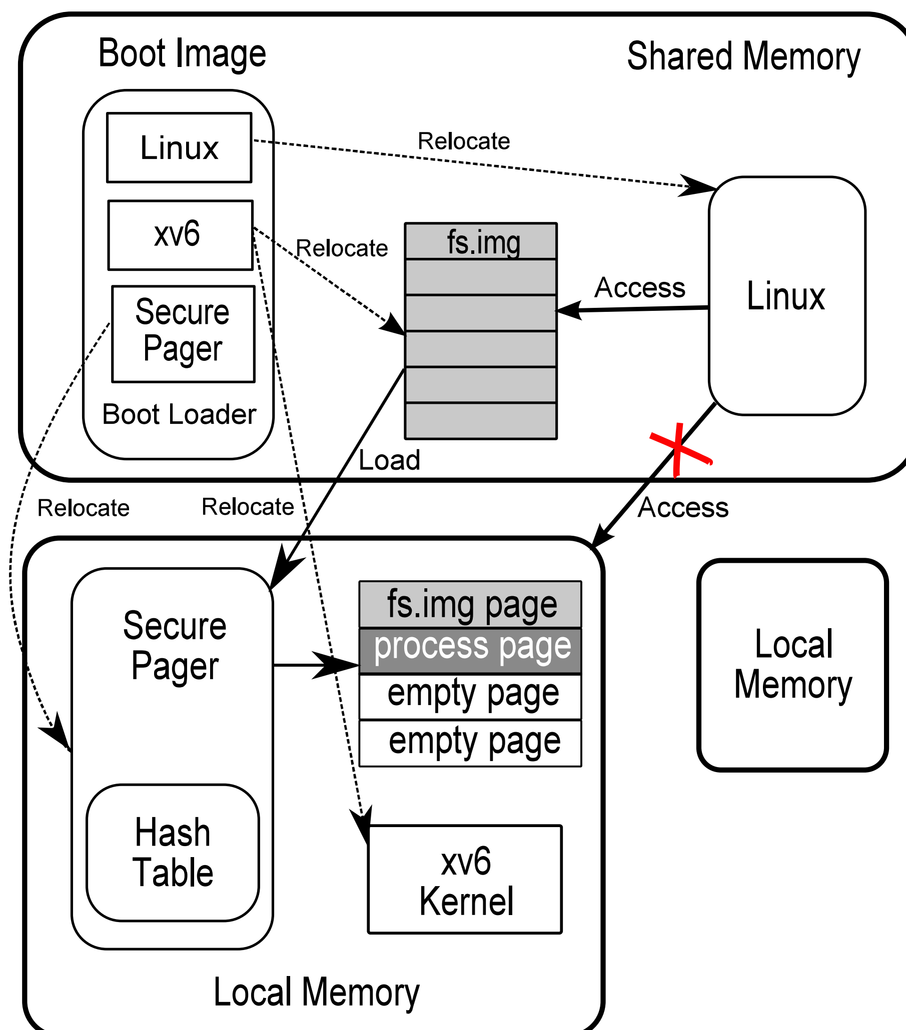

Figure 3.12: Optimized scheme.

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 integrity checker 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.

3.5 Evaluation

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

3.5.1 System Configuration

In order to evaluate our system, we use 3 kinds of configurations shown in Fig. 3.13. 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), there is only a Linux running on 3 cores.

Because the technology of the integrity checker is not the focus of this paper, we only apply limited functions into it as a sample integrity checker that can monitor the state of the embedded Linux. We consider that this is enough to evaluate our architecture effectively. We choose an integrity checker, which can check the kernel system call table of Linux and the *hide_task* rootkit, and calculate the execution time of the integrity checker to evaluate the overhead. The size of the integrity checker 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 the performance of our system with these configurations. Firstly, we want to make sure whether our system causes heavy overhead to the Linux, which is the most important in our architecture because it relates firmly to users that operate directly in the Linux.

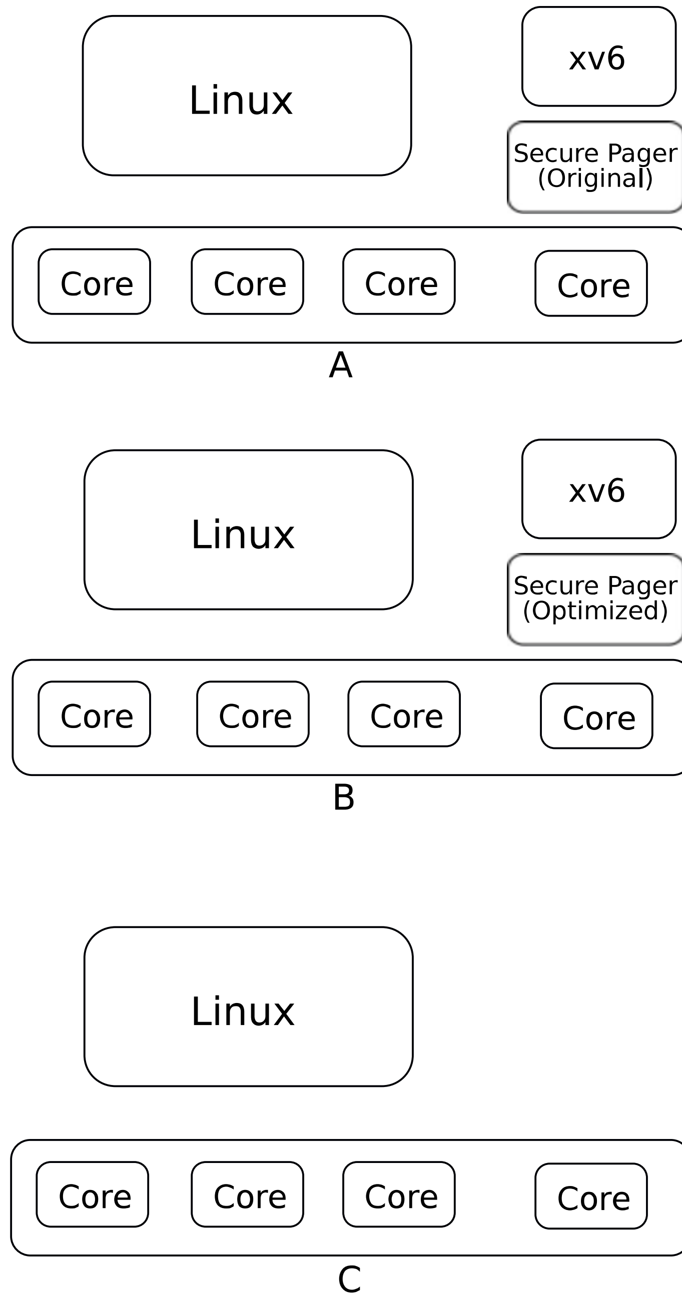

Figure 3.13: System configuration.

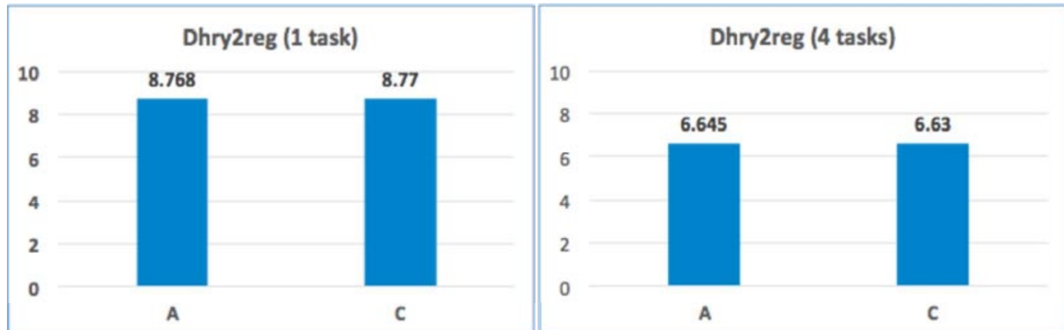

Figure 3.14: Dhry2reg results (10^6 loops).

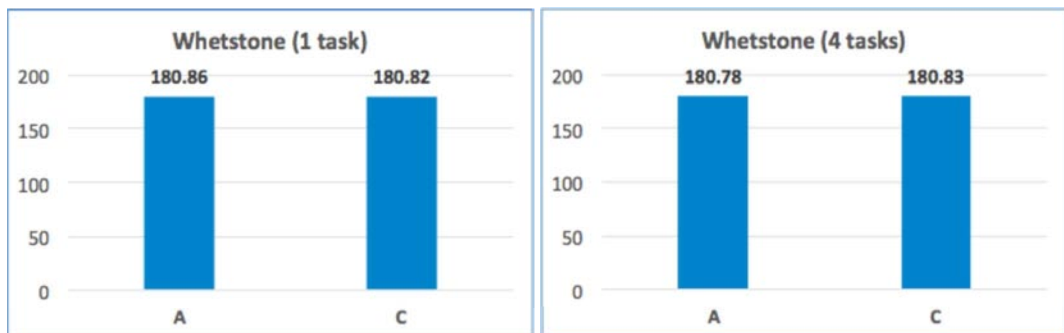

Figure 3.15: Whetstone results (MWIPS).

3.5.2 Performance of Linux

We verified whether the performance of Linux was influenced by our architecture. We ran several test tools from Unixbench [28] in configuration (A) and 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. 3.14, Fig. 3.15, Fig. 3.16.

Then we run the same tools in configuration (B) to make more verification and the result is shown in Fig. 3.17, Fig. 3.18, Fig. 3.19. It is predictable that we did not obtain results that are significantly different from results in configuration (A) and (C). The difference among them is minimal.

From these figures, we can argue that the performance of Linux is minimally

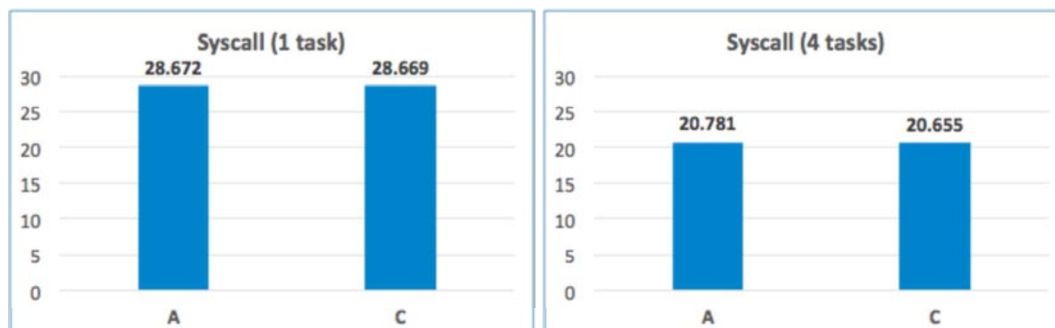

Figure 3.16: Syscall results (10^5 loops).

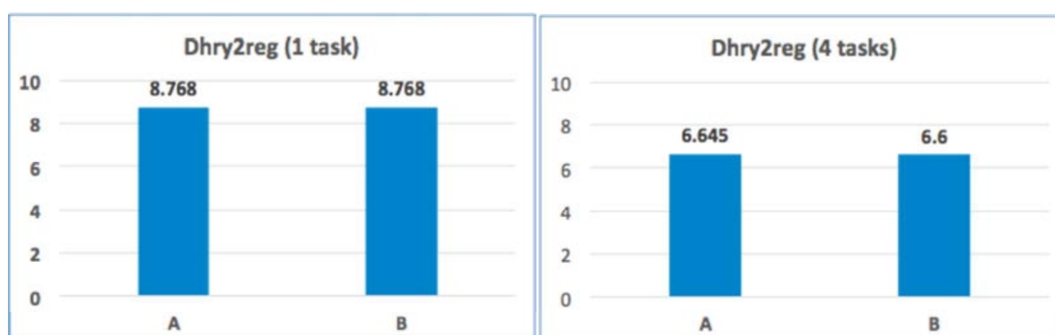

Figure 3.17: Dhry2reg results (10^6 loops).

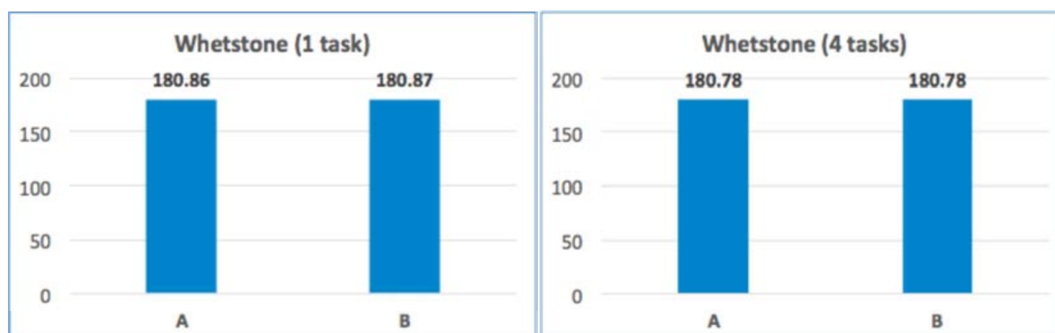

Figure 3.18: Whetstone results (MWIPS).

influenced by our architecture. We consider that the influence is from the passive integrity checker, which would occupy memory bus during runtime. The secure pager may increase the execution time of the integrity checker, which may lead to more bus time occupation. However, because the integrity checker 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.

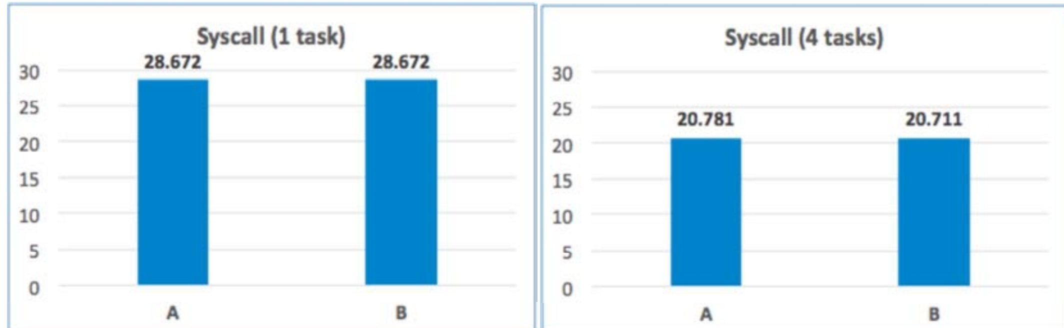

Figure 3.19: Syscall results (10^5 loops).

Because the difference between these configurations is very small, we consider that the random performance variation on the platform may have influence on it. This illustrates that a light integrity checker does not have a significant larger influence on the performance of Linux, which should be minimal in the embedded environments.

3.5.3 Integrity Check Function

We ran the integrity checker in 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 integrity checker by verifying the integrity when Linux runs. The whole reliability of the system is also improved.

3.5.4 Automatic Update Function

In order to validate the automatic update function, firstly we ran an integrity checker that could only check the kernel system 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.

Table 3.4: Decryption time of update file.

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

When the first module was not loaded in our experiment, the integrity checker 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 integrity checker.

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 integrity checker.

3.5.5 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 configuration (A): RSA, DES and a simple encryption mechanism, which only divides every page of the integrity checker 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 integrity checker with these 3 mechanisms in the host machine and did the update in RP1. The time listed in Table. 3.4 shows the decryption performance.

According to Table. 3.4, it shows that RSA takes much more time than DES

Table 3.5: Read attack protection overhead (ms).

Encryption Mechanism	Execution Time
None	94.3
RC4	172.9

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 integrity checker needs to be updated, does not influence the performance of the integrity checker during runtime and cannot be noticed by users. In real applications, a proper encryption mechanism is required to be chosen to suit configuration of embedded platforms to obtain a good balance between the security and the performance.

3.5.6 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 integrity checker and this overhead influences the integrity checker continually. According to Table. 3.4, complex encryption methods are unsuitable for embedded systems. We chose an encryption mechanism, stream cipher RC4 [45], to evaluate the overhead in configuration (A), and computed the execution time of the integrity checker with or without RC4 encryption.

The results shown in Table. 3.5 illustrate that this encryption mechanism introduces a reasonable overhead during the running of the integrity checker. The encryption mechanism should be applied depending on the hardware configuration to obtain a reasonable tradeoff between the security and the performance.

Table 3.6: Integrity checker evaluation.

Configuration	Execution Time (ms)	Page Swaps (times)
A	94.3	22
B	4.2	0

3.5.7 Comparison between Original and Optimized Schemes

In order to compare the original scheme and the optimized scheme, we ran the same integrity checker in configuration (A) and configuration (B), and the execution time of the integrity checker was calculated shown in Table. 3.6.

From Table. 3.6, it indicates that the execution time in configuration (B) is much shorter than in configuration (A) and there are 22 page swaps in configuration (A). We consider that the number of page swaps incurs this large difference. In the original scheme, both the *fs.img* (containing the integrity checker) and the xv6 kernel are stored in the shared memory. When the integrity checker uses system calls, pages of the xv6 kernel are swapped in and out to run the integrity checker. Due to page copy and hash calculation during the page swap, the speed of the integrity checker 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 the 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 configuration (A), we obtained the execution time of the integrity checker in Table. 3.7 and the copy time of each page (22 pages) is 0.5 ms (software method) or 0.005 ms (DTU). It shows that the DTU method is much faster than the software method, and the memory copy takes much time during the execution of the integrity checker.

Secondly, we used another hash algorithm MD5 in the secure pager with simpler complexity and less computing calculation than SHA1. We applied these

Table 3.7: Page copy overhead evaluation (ms).

Copy Type	Execution Time	Per Page (22 pages)
Software	94.3	0.5
DTU	83.7	0.005

Table 3.8: Hash calculation overhead evaluation (ms).

Hash Algorithm	Execution Time	Per Page (22 pages)
SHA1	83.7	3.4
MD5	33.8	1.1

two methods to the secure pager in configuration (A) to get the execution time of the integrity checker shown in Table. 3.8 and the computing time of each page (22 pages) is 3.4 ms (SHA1) or 1.1 ms (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 integrity checker. Although MD5 is not so robust in today's applications, the results show that a proper hash mechanism needs to be selected to provide a good tradeoff between the security and the performance.

We then introduce a large integrity checker into our system. To reduce the page tables required by this integrity checker, we split it into many smaller instances and execute these instances sequentially in our system, as shown in Fig. 3.20. We do not build a true large integrity checker, but we instead assume that the multiple integrity checkers used in our evaluation contains a large integrity checker. We run these instances sequentially in (A) and (B) for 20 times, equivalent to executing a large integrity checker of 160 Kbytes, and we only allowed one instance to be run concurrently.

The execution time of the integrity checker consists of two main parts: the loading time in which xv6 forks the user process and loads the small instances into the local memory and the execution time of the small instances, which is the same as in the above evaluations. We should recall that the loading time is part of the above evaluations. However, it occurs only once during the loading of the integrity checker, and the integrity checker performs its tasks repeatedly without reloading its own content. Therefore, we eliminated the loading time from the previous evaluations. In this evaluation, the loading time would be much larger

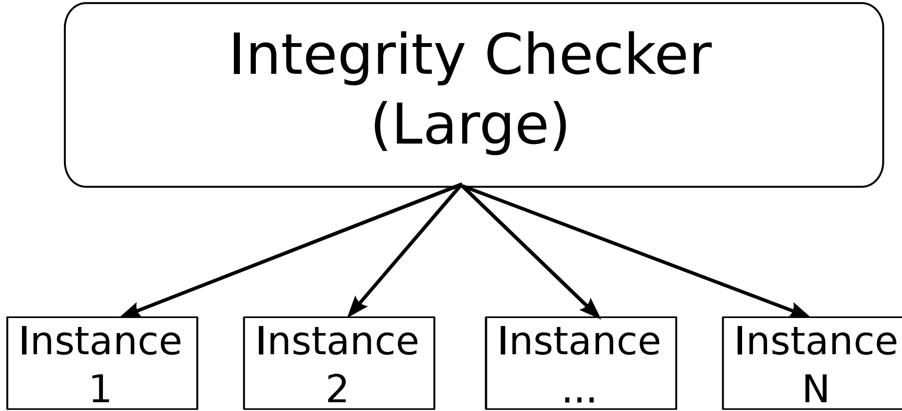

Figure 3.20: Split large integrity checkers.

because every time an instance is executed, the content of the instance must be loaded. We calculated the total execution time from the start to the end of the execution as well as the execution time of each single instance of running the checking tasks.

The total execution time of the large integrity checker is shown in Table. 3.9 for configuration (A) and in Table. 3.10 in configuration (B). The execution time for each instance of running the checking tasks is shown in Table. 3.11 for configuration (A) and in Table. 3.12 for configuration (B).

Regarding the total execution time, configuration (B) is much faster than configuration (A) due to the much smaller number of page swaps, which influences the performance of an instance significantly. Based on the execution time for each individual instance, we argue that in configuration (A), the pages are not loaded according to a fixed pattern for each instance or many instances. The execution time varies from 12.4 ms to 86.1 ms and the number of page swaps varies from 4 to 22. The average execution time is 53.8 ms, and the average number of page swaps is 14.1. In contrast, in configuration (B), the execution time of each instance is constant (4.2 ms) with a single 8 Kbytes integrity checker.

The secure pager loads into the required pages for the instance. In configuration (A), because the pages in the local memory are swapped out randomly, the number of page swaps is flexible for different instances, which increase the overhead. In configuration (B), it is unsurprising that the execution time of each

Table 3.9: Total execution time of the large integrity checker in configuration (A).

	Execution Time (ms)
Total	5882.4
Average	294.1

Table 3.10: Total execution time of the large integrity checker in configuration (B).

	Execution Time (ms)
Total	269.5
Average	13.5

Table 3.11: Single instance execution time in configuration (A).

Time	Execution Time (ms)	Page Swap (times)
1	53.4	14
2	53.4	14
3	69.8	18
4	20.6	6
5	86.1	22
6	37.0	10
7	37.0	10
8	78.0	20
9	20.6	6
10	77.9	20
11	78.0	20
12	12.5	4
13	86.1	22
14	28.8	8
15	45.2	12
16	86.1	22
17	45.2	12
18	86.1	22
19	28.8	8
20	45.2	12
Average	53.8	14.1

instance remains almost the same. During the runtime of the integrity checker in configuration (B), no page swaps are required, and the performance of the

Table 3.12: Single instance execution time in configuration (B).

Time	Execution Time (ms)	Page Swap (times)
1	4.2	0
2	4.2	0
3	4.2	0
4	4.2	0
5	4.2	0
...	...	...
16	4.2	0
17	4.2	0
18	4.2	0
19	4.2	0
20	4.2	0
Average	4.2	0

integrity checker is not affected.

When we just want to run a small integrity checker, we only need to load in it for the first time and execute it circularly. However, if we execute a large integrity checker by running small instances, we must load in one instance, execute it, exit, clear its memory space and begin to load in the next instance. Each instance requires such a procedure, therefore, the execution of the instance only takes part of the whole execution time and other parts consume most part of the whole execution time. We can find that this method brings in great overhead to the performance of the integrity checker. However, with this method, our architecture can also be applied on large integrity checkers to extend the application.

3.5.8 Future Analysis of the Influence of the Secure Pager

In the optimized scheme, we found that placing the kernel part entirely in the local memory could avoid page swaps during the runtime of the integrity checker, which improves the performance of our architecture significantly. We then perform a more detailed analysis to improve the performance of the integrity checker. We made some modifications on configuration (B), which could only place *.start* and *.bss* sections permanently into the local memory and other parts into the shared

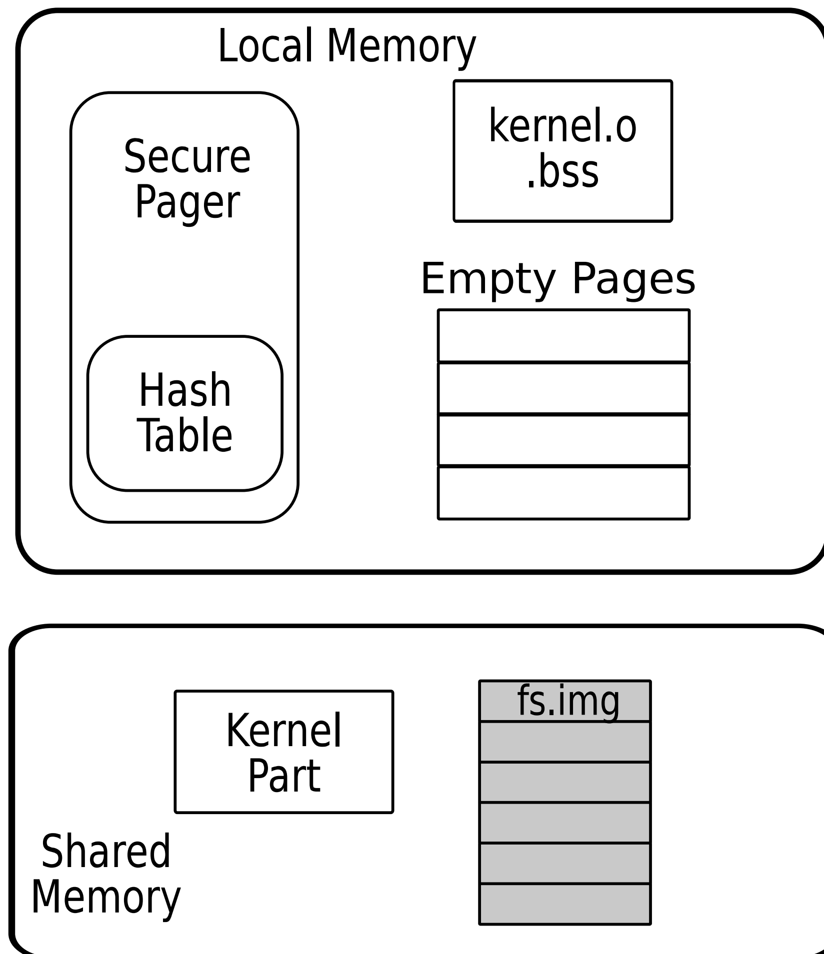

Figure 3.21: *kernel.o* is located permanently in local memory in configuration (A).

memory, as shown in Fig. 3.21. We then calculated the execution time for each instance and for the large integrity checker shown in Table. 3.13 and Table. 3.14.

We can find that the execution speed is much slower in this modification than in configuration (B) and similar to configuration (A). In fact, this system configuration is more similar to configuration (A) than to configuration (B). Therefore, the page-swap situation is somewhat similar to the situation in configuration (A). However, the overall time required by the integrity checker is much slower than in configuration (B) and still much faster than in configuration (A).

Because *kernel.o* contains the critical system call route and interrupt handler

Table 3.13: Total execution time of the large integrity checker with *kernel.o* located in local memory in configuration (A).

	Execution Time (ms)
Total	761.5
Average	38.1

Table 3.14: Single instance execution time with *proc.o* located in local memory.

Time	Execution Time (ms)
1	85.0
2	85.0
3	85.0
4	85.0
5	85.0
...	...
16	85.0
17	85.0
18	85.0
19	85.0
20	85.0
Average	85.0

route, it can help reduce the number of page swaps during the loading of the instances if it is located in the local memory. Therefore, the total execution time is faster in this configuration than it is in configuration (A). However, this change alone does not influence the page swap situation significantly during the runtime of each instance, and thus their execution times are similar to configuration (A).

Then, we moved the *proc.o*, which completed the operations regarding the user processes, such as scheduling, sleeping, waking and waiting, into the local memory, as shown in Fig. 3.22, and recalculated the execution time of each instances and the large integrity checker.

The results in Table. 3.15 and Table. 3.16 show that the execution time of each instances is 4.2 ms, which is nearly equal to the execution time in configuration (B), and indicates that during no page swap is needed during the execution of each instance. These results indicate that even small changes to the kernel may change the page swap situation during the execution of each instances. However,

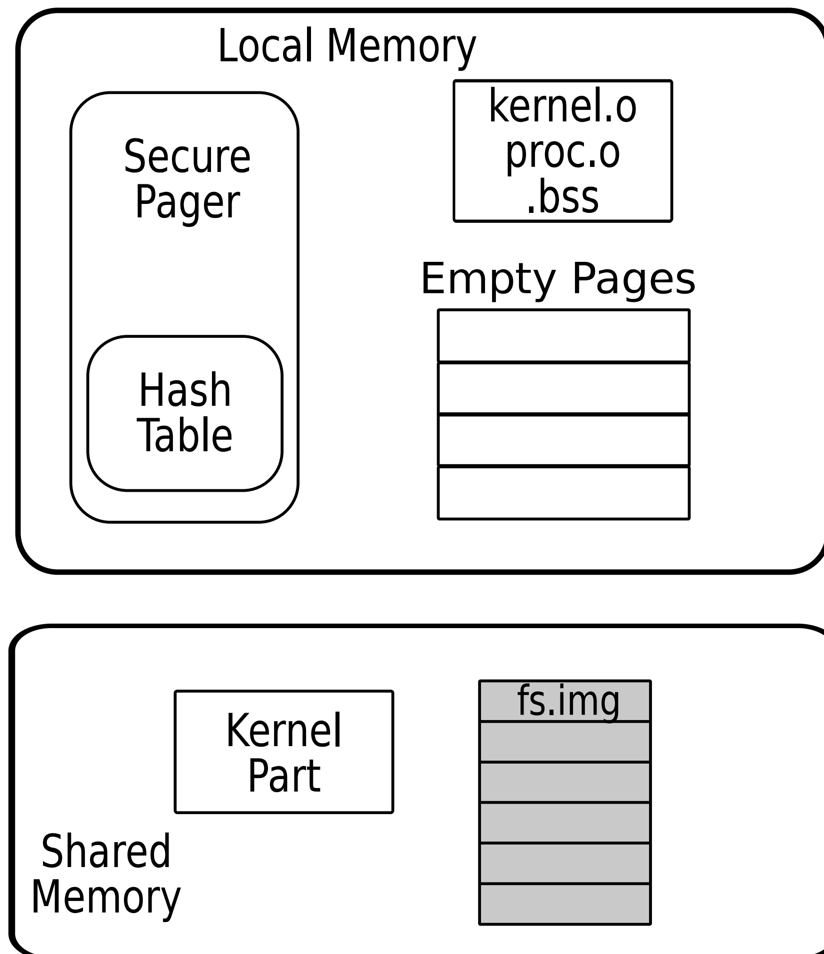

Figure 3.22: Add *proc.o* into local memory.

the total execution time of the integrity checker is similar (764.5 ms and 764.8 ms) to the previous evaluation, which was configured with only *kernel.o* located in the local memory, thus showing that even if the number of page swaps for each instances is greatly reduced, the number of page swaps during the execution of the whole integrity checker is changed minimally. This evaluation shows that modifications to the kernel relocation are sensitive and the effect is difficult to predict.

Next, we performed some similar evaluations with the relocation of the kernel of xv6. One case involved placing *console.o* into the local memory together with *kernel.o*, as shown in Fig. 3.23, and another added *scif.o* into the local memory, as

Table 3.15: Total execution time of the large integrity checker with *proc.o* located in local memory.

	Execution Time (ms)
Total	767.8
Average	38.4

Table 3.16: Single instance execution time with *kernel.o* located in local memory.

Time	Execution Time (ms)
1	4.2
2	4.2
3	4.2
4	4.2
5	4.2
...	...
16	4.2
17	4.2
18	4.2
19	4.2
20	4.2
Average	4.2

Table 3.17: Total execution time of the large integrity checker with *console.o* or *console.o* + *scif.o* located in local memory.

Execution Time (ms)	*console.o*	*console.o* + *scif.o*
Total	891.0	884.4
Average	44.6	42.2

shown in Fig. 3.24. The execution time for each instance and the large integrity checker were calculated for these two cases, and the results are shown in the Table. 3.17 and Table. 3.18.

The execution times of each instances are a little slower than in configuration (B); only a few page swaps are needed during the runtime of each instance. However, we find that the total execution time is not changed according to a fixed rule. Adding *scif.o* can slightly reduce the execution time of the whole integrity checker, but both configurations take longer than configuration that only places *kernel.o* into the local memory.

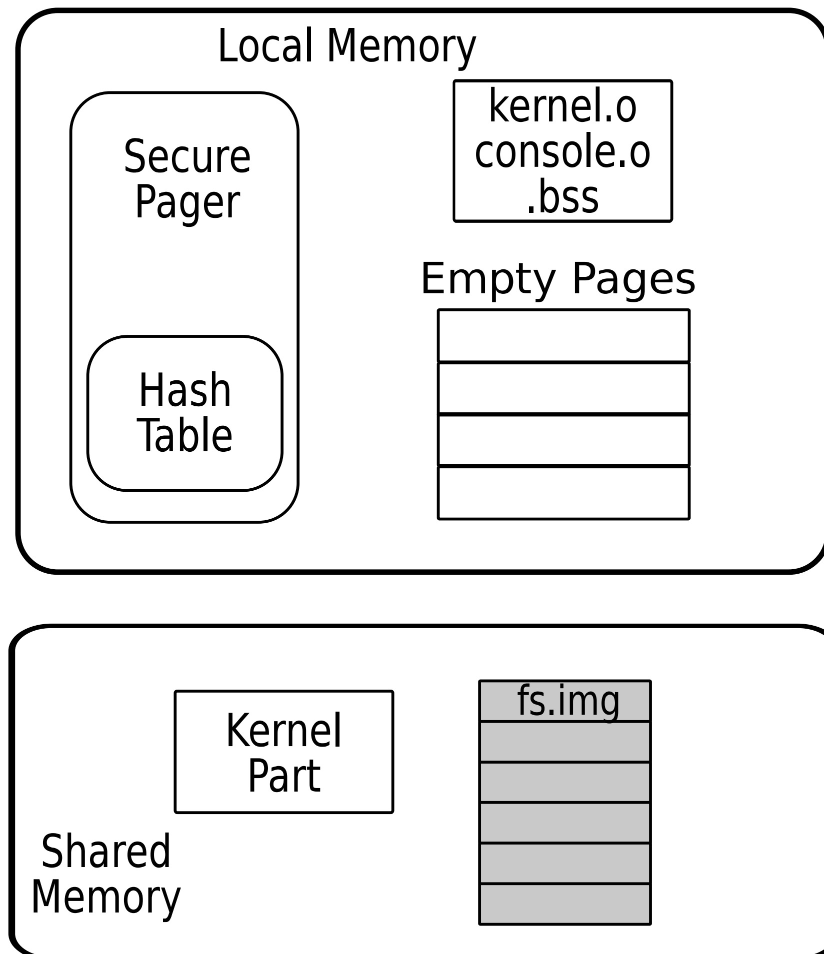

Figure 3.23: Add *console.o* into local memory.

The page usage situation during the loading and execution of the instances can be very complex. Therefore, placing only one *.o* file into the local memory may reduce the number of page swaps during the runtime of each instance, but it does not significantly influence the page usage situation throughout the execution of the large integrity checker. Conversely, the *.o* occupies space in the local memory and would therefore reduce the space in the local memory for the content of the instances. Especially in our evaluations, because we manage the local memory using 4 Kbytes pages, the *.o* still uses a full page even if it is smaller than 4 Kbytes, which would waste valuable local memory space. We argue that this issue also contributes to the low speed of the integrity checker with small instances.

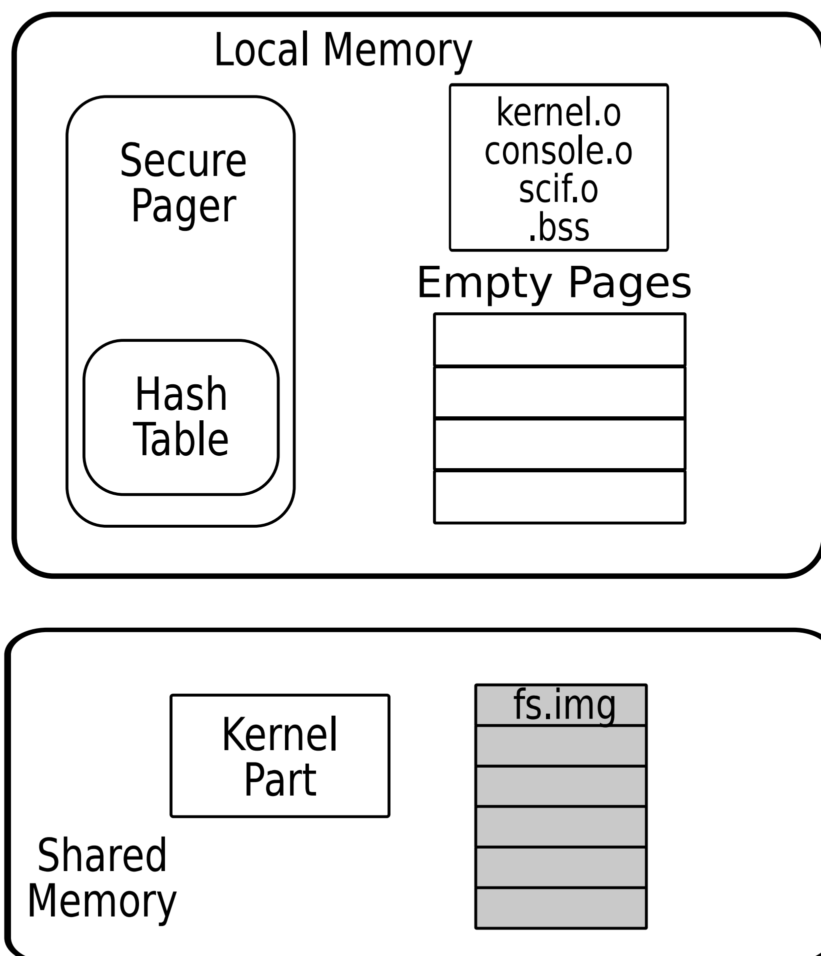

Figure 3.24: Add *console.o* and *scif.o* into local memory.

To verify this reason. we placed random *.o* files into the local memory and obtained a situation that significantly harmed the overall performance of the large integrity checker. This special configuration is shown in Fig. 3.25, with *ide.o exec.o* and *scif.o* placed into the local memory, and the execution time of each instances and the entire integrity checker are shown in Table. 3.20 and Table. 3.19. The total execution time of the integrity checker is much slower than in the other configurations, which shows that relocating the xv6 kernel does not always speed up the execution of each instances, and the entire integrity checker. When we want to load parts of the xv6 kernel into the local memory and never swap out, we should consider about the tradeoff between the space

Table 3.18: Single instance execution time with *console.o* or *console.o* + *scif.o* located in local memory.

Execution Time (ms)	*console.o*	*console.o* + *scif.o*
1	5.0	5.0
2	5.0	5.0
3	5.0	5.0
4	5.0	5.0
5	5.0	5.0
...	...	...
16	5.0	5.0
17	5.0	5.0
18	5.0	5.0
19	5.0	5.0
20	5.0	5.0
Average	5.0	5.0

Table 3.19: Total execution time of the large integrity checker with a bad case.

	Execution Time (ms)
Total	24965.6
Average	1248.3

for storing the swap content and the space for storing the permanent content of the kernel. We should place the most commonly used pages into the local memory, which reduces the number of page swaps during the whole execution or the runtime of the instances. This type of kernel relocation can improve the execution speed of each instances, but may not reduce the number of page swaps during the overall execution of the integrity checker. We also predict that if the checking functions in the integrity checker changes, the page swap situation may also change significantly during the running of the integrity checker. Once we make some optimizations on the execution of the integrity checker, careful experiments should be performed to verify the improvement in the performance. A complex kernel execution situation may incur some unexpected page swaps that harm the performance optimization of our configuration.

Next, we examined how the size of the spare space in the local memory influences the integrity checker. We reduced the spare space in the local memory

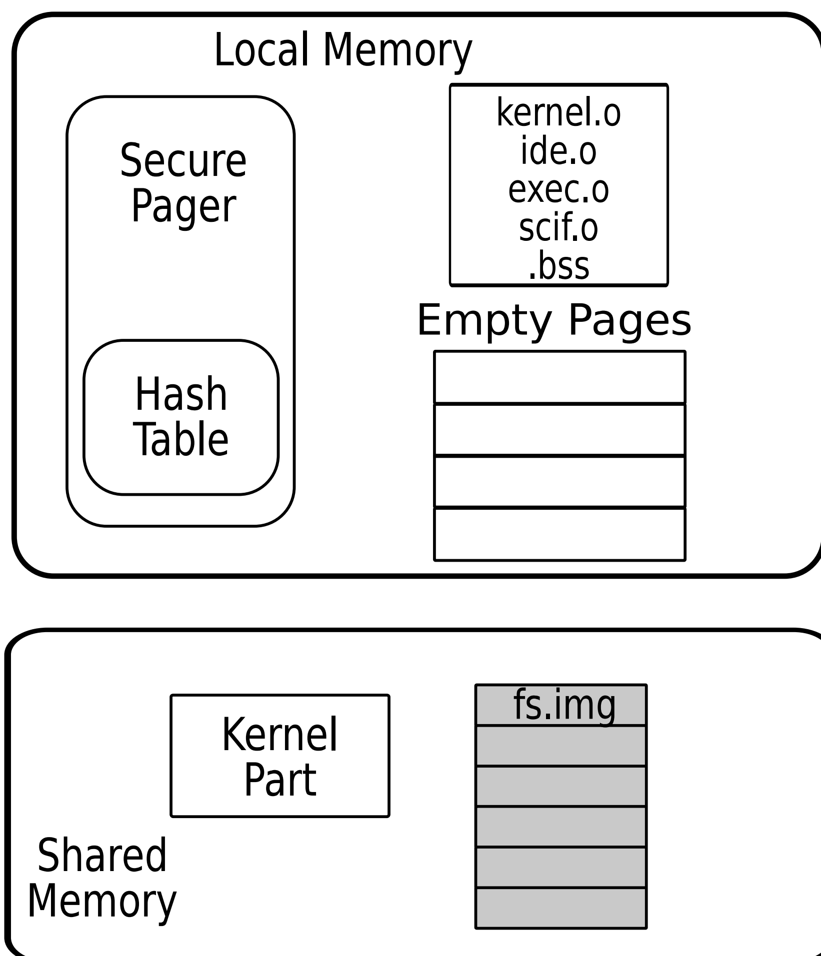

Figure 3.25: A bad case of relocation of the kernel.

for swapping in the integrity checker in configuration (A) with a single integrity checker. Initially, there was 76 Kbytes (19 pages) of space in the local memory to execute the integrity checker. We reduced this value in increments of 4 Kbytes (1 page) and calculated the execution time of the integrity checker. The results are shown in the Table. 3.21.

We can find that when the spare space is reduced from 19 pages to 18 or 17 pages, the execution time of the integrity checker does not change very much, but remains in a stable state throughtout the runtime of the integrity checker. However, when the number of the spare pages is reduced to 15, the execution time of the integrity checker rises to a hyperbolic value, which is not the same

Table 3.20: Single instance execution time with a bad case.

Time	Execution Time (ms)
1	85.5
2	85.5
3	85.5
4	85.5
5	85.5
...	...
16	85.5
17	85.5
18	85.5
19	85.5
20	85.5
Average	85.5

level observed in the previous cases. We argue that the size of the spare space in the local memory is not sensitive to the page swaps during the runtime of the integrity checker. Up to a certain point, it can loads in the required pages for the integrity checker, and maintain a similar execution time for the integrity checker in a similar level. However, if the spare space in the local memory is reduced sufficiently, the page swaps for the execution of the integrity checker would increase enough to significantly influence the performance of the integrity checker. In other words, if we can maintain the desired level of spare space in the local memory, we can achieve an acceptable performance for the integrity checker. If there is enough space in the local memory for the entire monitor OS, the performance is optimal because there is no need for a paging mechanism with a secure pager and there is no extra overhead during the runtime of the integrity checker. However, the local memory is rarely this large. We still need to determine that how to optimize the execution of an integrity checker for a reasonable speed.

The page usage of an OS is very complex, and it is thus difficult to make absolute judgments and rules on how to design an integrity checker within good performance. Many elements can influence the execution of an integrity checker, and one element can influence another during runtime. For example, we executed

Table 3.21: Execution time in configuration (A) with reduction of spare space.

Time	19 pages (ms)	18 pages (ms)	17 pages	16 pages (ms)
1	69.8	94.3	94.3	4810.8
2	94.3	94.3	94.3	4516.3
3	86.2	94.3	94.3	4516.3
4	86.2	94.3	94.3	4516.3
5	86.2	94.3	94.3	4516.3
6	94.3	94.3	94.3	4516.3
7	94.3	94.3	94.3	4516.3
...	...	...	...	...
16	94.3	94.3	94.3	4516.3
17	94.3	94.3	94.3	4516.3
18	94.3	94.3	94.3	4516.3
19	94.3	94.3	94.3	4516.3
20	94.3	94.3	94.3	4516.3
Average	91.7	94.3	94.3	4531.0

the integrity checker 1 times/s with a *sleep()* system call in configuration (A). However, we also removed this system call and recalculated the execution time of the integrity checker, as shown in Table. 3.22.

We find that the second time we execute the integrity checker, the execution time becomes very different, and the integrity checker runs much faster without sleeping. This result verifies that the *sleep()* function would swap out some pages that are required during the runtime of the integrity checker. To execute the integrity checker again, these pages must therefore be swapped into the local memory, which reduces the performance of the integrity checker. Thus even simple modifications to the integrity checker might significantly affect its performance.

Finally, we wanted to find a way to optimize the performance of the integrity checker. We tried to log the number of times that certain critical functions were called during the runtime of the integrity checker. Through analysis of the xv6 kernel, it can be determined that some functions are called frequently during runtime to perform certain actions such as file reading and loading or process scheduling. We added counters to many functions to obtain the number of times these functions were called during the runtime in configuration (A), as shown in

Table 3.22: Execution time in configuration (A) with or without a *sleep()* function.

Execution Time (ms)	with *sleep()*	without *sleep()*
1	69.8	69.8
2	94.3	5.0
3	86.2	5.0
4	86.2	5.0
5	86.2	5.0
6	94.3	5.0
7	94.3	5.0
8	94.3	5.0
...	...	...
16	94.3	5.0
17	94.3	5.0
18	94.3	5.0
19	94.3	5.0
20	94.3	5.0
Average	91.7	8.24

Table. 3.23. Based on the results, we can determine which files are used more frequently, and should therefore be permanently located in the local memory at a higher privilege level. It can also be determined that a single integrity checker and a large integrity checker with many instances differ differ greatly in the number of times that many functions are called.

Two factors should be considered in this evaluation. First, the counters or other debug codes may influence the page swaps during the runtime of the integrity checker; therefore, the performance of the integrity checker when we try to obtain the counter number may not be accurate without these debug codes. Second, as shown in Table. 3.23, the page usage situation may change greatly when we redesign the integrity checker. Therefore we must perform this evaluation every time we apply a new integrity checker for optimization. However, for a specific integrity checker, the number of times that the functions are called can help us determine which files are more frequently required by the integrity checker and therefore should have a preferred placement in the local memory without swapping out. This approach is more likely to improve the performance

Table 3.23: Execution times of functions in configuration (A) (sample).

Function Name	File Name	Integrity Checker With Instances	Single Integrity Checker
swtch()	*proc.o*	121	35
wait()	*proc.o*	20	1
exec()	*exec.o*	21	2
fork()	*proc.o*	20	1
consolewrite()	*console.o*	1118	1118
ide_start()	*ide.o*	378	36
yield()	*proc.o*	5	0
sleep()	*proc.o*	35	32
wakeup()	*proc.o*	40	32
kill()	*proc.o*	0	0
exit()	*proc.o*	20	1

of an integrity checker than it is to harm its execution.

We also consider another method of optimizing the execution of the integrity checker; redesigning the monitor OS kernel. We can divide the kernel into many pages and place files that contains corresponding functions into each page. We prefer to place the most frequently required functions into one page so that the integrity checker does not require other pages to be loaded into the local memory for execution during runtime. However, this method has three issues. First, it is difficult to make one file or the sum of multiple files exactly fit the size of one page or several pages (4 Kbytes or 4*[N] Kbytes). This method would therefore waste some spaces when the size of a file or files is not an exact multiple of 4 Kbytes. Second, substantial design effort is required. We have to consider the size of the file when implementing the functions, which adds considerable extra effort. Finally, if we want to add new functions or optimizations, the size handling makes the process tedious with significant development time. We prefer not to modify the monitor OS in ways that would require extensive design effort and therefore consider this method to be unsuitable for a project that may experience much nondeterminacy during development.

Based on the above evaluation, we can make some suggestions regarding how to improve the performance of an integrity checker in the LLM architecture, in

addition to using more powerful processors.

- Use system calls carefully in the integrity checker. The system calls access kernel pages, which may incur page swaps.
- Place the most frequently used pages in the local memory and never swap them out. This method can efficiently decrease the number of page swaps and thus reduce overhead.
- Optimize the integrity checker based on an accurate understanding of the page usage situation in the kernel. Not every attempt to optimize the integrity checker has a positive effect on its performance of the integrity checker. The page usage situation in the kernel is very complex; therefore, an accurate understanding situation of the page usage plays as a key role in any optimizations.
- Use hardware to transfer pages and calculate hash values. Generally, hardware computation is faster than software computation, and the CPU can address other workloads during the hardware computation.

4

Discussion

Our approach replies on several hardware processor features to improve the security or reduce the size requirements for the local memory. In this chapter, we first introduce many studies regarding these features, including hypervisor support on embedded processors and software cache&TLB architecture. Next we analyze our approach and discuss whether these features can be equipped on future processors. Finally, we discuss our current research and propose some hardware recommendations to generalize the LLM machine architecture.

4.1 Hardware Virtualization Support in Embedded Processors

Hypervisors are currently being introduced into embedded systems. The inspiration for embedded hypervisors is not the same as the reasoning for traditional hypervisors. Embedded systems commonly contain many critical or real-time tasks along with more general tasks, and embedded hypervisors are expected to manage them with the appropriate privileges and isolation. For example, in smart phones, the mobile talk service and the 2G/3G communication service are critical tasks that are basic functions of a phone and should be managed with high privilege and security. Conversely, web survey and multi-media services are

general purpose tasks that do not need to be managed so carefully. With embedded hypervisors, we can run a real-time or a secure OS and a general-purpose OS concurrently to execute tasks with the appropriate privilege and security. Although many studies [30] [48] [57] have tried to build multiple OS environments on embedded processors, they should consider about the performance and design effort required to apply their methods.

While main-stream desktop and server processors such as Intel and AMD have already been equipped with some level of virtualization extensions, virtualization support is not a common feature in embedded processors. Most ARM processors, which dominate the embedded market, do not incorporate hardware virtualization extensions. However, the latest ARM processors, Cortex A15 [1] [27], have been announced as being equipped with virtualization extensions. Intel, which has advanced virtualization technologies in its hardware and software, has already produced many desktop and server processors with hardware virtualization and has begun to equip its high-end embedded processor with such features. With improvements in the performance and manufacturing processes, hypervisors might become a necessary feature of embedded platforms and hardware virtualization extensions may become a standard configuration in future embedded processors. However, at the same time, many embedded applications remain sensitive to cost and do not require hardware virtualization extensions. Adding virtualization extensions to these embedded processors would be a waste of resources. We foresee that future embedded platforms, there should contain both embedded processors both with and without virtualization extensions depending on their applications.

However, our approach does not aim to replace hypervisors and can coexist with a hypervisor. This approach can be applied as a security enhancement for hypervisors using an LLM-similar architecture processor, and we can incorporate checking technologies into the integrity checker that can check the target OS and the hypervisor simultaneously. Our approach also provides a good low-overhead security solution for embedded systems without importing hypervisors.

The development of hypervisors can apply pressure toward the hardware development process. Researchers find many issues when they implement a hypervisor on an embedded device, and they must consider both software and hardware

methods to solve them. Because the hardware features are not specific to these solutions, potential hardware improvement are usually found when the solutions are evaluated, and the performance overhead and design effort can be reduced significantly using suitable hardware features. After ARM announced the virtualization support in its Cortex A15, several projects were proposed to utilize these features in implementing hypervisors. These hardware features would also promote the development of further software technologies.

We now investigate the development of other features equipped in processors, such as the hardware-managed TLB and the hardware-coherency cache.

4.2 TLB Management

Hardware-managed translation lookaside buffer (TLB) is commonly applied in the modern x86 processors and ARM processors. To establish TLB automatically in the processor hardware, the format of the page tables of the guest OS must be fixed. When page faults do occur, the processor can then search the page tables of the system and set the corresponding address translation in TLB. However, this approach introduces some problems, in that it is very difficult to add extra features to page tables if the processor is not equipped with this feature, and fixed-format page tables would restrict the application of hardware-managed TLB in size-critical cases. Other modern processors prefer to use software-managed TLBs, such as MIPS, Alpha, SPAC, and PA-RISC, which can provide flexibility in the format of page tables that can improve the performance of many applications [31] [53] [65] [72].

The research proposed by Asmer Jaleel and Bruce Jacob [31] uses a novel method based on the reorder buffer to improve the performance of precisely handling of software-managed interrupts. It focuses on the most frequently occurring TLB miss interrupts, which may incur noticeable overhead for the overall system performance, and uses the reorder buffer to store frequently accessed data to avoid reloading and re-executing. The evaluation shows that this method can avoid 50%-90% of TLB interrupts and improve the performance of the system by 5-25%.

Torong & Day applied an imprecise-interrupt mechanism [65] to handle interrupts that are transparent to application program semantics. It can stores the instruction data form the reordered buffer when an interrupt is trapped and can selectively restore them when the interrupt exits. The evaluation shows that this approach is more suitable for a heavy interrupt handler, and is less useful for recently TLB interrupts with relatively low overhead, for which this approach may require more processor operations.

Jinzhan Peng et al. applied many hardware/hardware combined TLB algorithms for execute Java applications [53] and compare their performance. They test many methods of reducing TLB miss rates and TLB miss latency and even include a NO-TLB design to remove the TLB overhead. The experimental results illustrate that there is a consistent performance bottleneck with the TLB, and the hardware TLB mechanisms have different performance implications. The software method including the NO-TLB design is somewhat suitable for executing a Java application. Bothe hardware-managed and software-managed TLB have their own pros and cons, and developers should determine to how to build a system based on specific situations.

Based on the above, the hardware-managed TLB and software-managed TLB are useful for different applications and none of them have a fatal weak point. We can choose suitable TLB algorithms depending on the specific application. To meet various requirements in the modern computing environment, we expect that both of these features would exist in the main-stream processors.

4.3 Cache Management

A cache is a high-access-speed memory region equipped to a processor that is used to adapt the difference between the extra high computing speed of the processors and the low access speed of the main memory. The cache can load the content of the main memory for the processor access to improve the overall throughput of the whole systems. The cache is usually managed by processors and is transparent to the software, so that when we develop a software application, there is no need to consider the cache utilization. In a desktop or server, the hardware

cache can improve the overall performance of the system. However, a hardware cache is not always suitable for embedded systems, especially in real-time systems, because the predictability of tasks in a real-time system is very important, and unexpected cache misses may influence the predictability significantly. In traditional solutions, people prefer to disable the cache mechanism in processors or use processors without a cache to achieve absolute predictability for real-time tasks, which sacrifices the system throughput performance. Furthermore, in a multi-core system, we must consider about cache coherency. Cache-coherency means that when many cores want to access the same memory block, the hardware should ensure that they would access the latest values in the block. If one core uses stall values, the whole system may run in an abnormal state. In a desktop or server processor, where the number of cores armed to a single processors is increasing, developing a hardware-coherency cache requires a complex hardware mechanism and immense design effort, although some projects have tried to implement scalable hardware-dependent cache-coherency protocols [19] [38] [64]. Other studies have maintained that in many-core processors, the hardware design for cache coherency becomes extremely difficult, and that using a software-coherency cache is thus unavoidable in many-core or scalable systems [12] [51]. To solve these problems, a software cache including software coherency is proposed to manage the system cache [14] [20] [46] [62] [67]. The software cache mainly optimizes task behavior to improve the system throughput and can be implemented either in the compiling phase [46] or in the running phase of the system kernel [20]. Both approaches increase the software design effort required to implement an applications by selectively loading the needed content into the cache during runtime. We do not take a strong position on which type of cache coherency is better but consider it to be a reasonable trade-off to move the design complexity from the hardware (hardware cache) to software (software cache).

One approach [46] implements compiler support for the software cache to improve the predictability of a preemptive system. It uses a software-based cache partition to control the behavior of tasks and associate distinct portions with real-time tasks. The compiler takes responsibility for partitioning the instructions and codes for tasks for cache utilization. It can improve the response times of tasks

after context-switch, but it produces overhead through cache reduction for each task and the additional instructions for data partition.

Another approach [14] using on the Cell-BE processor also tries to implement a software cache depending on a compiler. It implements a runtime library that can help the compiler to generate code to maximize the chances for overlapping communication and computation. It introduces basic code generation mechanisms in the compiler and optimization methods. The evaluations show a significant improvement in the performance against the Stream and Random Access benchmark.

Software cache methods can be included to improve the performance of specific tasks [20] [62]. The study [20] applies a software method to a scalable and efficient implementation of OpenMP on the IBM CELL architecture to help improve the performance. The study[62] provides an enhancement for the sum-products on Graphics Processing Units (GPUs). It designs memory-bound algorithms to allow high data reuse using the software-managed memory equipped on GPUs. The evaluations show a greatly improved speed of random data access and a significant improvement in performance on real-life genetic analysis datasets.

The possibility can still be raised that the development of a software-coherency cache may downgrade the overall throughput of the system or require more effort in task design. To achieve a good balance, some hardware support is proposed to help implement a software-coherency cache more easily and with better performance [13]. A novel hybrid hardware-software coherency mechanism is proposed, in which the software is responsible for trapping the coherency actions including self-invalidations and writebacks, and the hardware applies Bloom filters to perform more selective self-invalidation. The experimental results shows that against some benchmarks there is only a low performance downgrade, but some cases show severe performance penalties making this approach unviable for real applications. However, the selective invalidation in this approach can achieve as much as a 93% improvement in the worst-case performance and offers similar performance to full-hardware cache coherency.

We find that the issues in using a hardware-coherency cache encourage the development of software coherency and that the issues in software-coherency in

turn encourage the optimization of the hardware cache architecture. The hybrid cache architecture can then provide a reasonable balance between the design effort applied to hardware and software and system throughput.

4.4 Hardware Features in Our Approach

We use local memory that is similar to a software cache to execute our monitor OS and use core isolation to provide security for the execution of the monitor OS, which improves the overall security of our system. A software-coherency cache has already been equipped in many embedded processors, such as Coldfire MCF5249, PowerPC 440, MPC5554, ARM 940, ARM 946E-S [16] and AT91M40400 [55]. Core isolation, however, is not a common feature in modern processors, either in embedded fields or in desktop and server fields. We hope that our approach suggests that an isolated privileged core would be a reasonable feature for enhancing the security on a multi-core processor and that our implementation can illustrate that with modest hardware support and design effort, the construction of our architecture is a realistic solution for embedded systems.

Our approach also exploits the software-managed TLB in the RP1 to reduce the size of the page tables for the monitor OS. Software-managed TLB are also equipped on many processors for various purposes. Our approach introduces more ways to utilize the software cache and software-managed TLB, which is expected to add to the demand for a processor with these features. We also use a software paging mechanism to further extend the local memory to execute a fully functional monitor OS. Similar to the hybrid cache architecture design, the hybrid design in our system can provide enhanced security to embedded systems without significant design effort.

As discussed above, there are many types of processors around the world. Proposing a general technology suitable for most of processors is very important to the application of this technology, as general suitability can reduce the design effort of porting it to other processors. However, there are many processor manufacturers in today's market. It is difficult to equip all processors with truly similar hardware, which is also not good for competition and future development. We

consider that the extreme optimization of a solution for a specialized hardware choice is also a good approach for improving the performance or reducing the design effort. In real cases, we should apply the best technologies without the concern for a generic hardware feature. After solving problems, we should consider whether the proposed method can be generalized or whether the restrictions are excessive.

4.5 Discussion and Hardware Recommendation

We have evaluated and analyzed some factors that may influence the performance of our architecture. To conclude, our architecture and secure pager provide functions of integrity verification and the automatic update of the integrity checker without introducing excessive overhead in Linux. The encryption mechanism, the page copy method and the hash algorithm must be chosen carefully for good performance on embedded platforms.

We should note that devoting one core of the processor to security in our architecture may lead to resource waste in embedded systems. However, we also foresee that using a special core to enhance the reliability would be reasonable when the number of cores in one processor increases to 16 or more. Furthermore, there is a type of processors called heterogeneous multi-core architecture processors [35, 74], in which each core can have its own individual configuration for different workloads to provide a good balance between the performance and resource consumption. We can configure a single core with an appropriate configuration to execute the secure pager and the integrity checker, obtaining a reasonable tradeoff between resource consumption and security. However, we prefer software-managed TLB architecture for flexibly designing the page tables of the monitor OS, which can efficiently reduce the size of the secure pager. In hardware-managed TLB architecture, more design effort may be required to restrict the size of the page tables, which we leave for future work.

Based on the above discussion, we present some hardware recommendations that may make the LLM architecture more applicable to real processors.

- An additional monitor core with a local memory, as shown in Fig. 4.1. This core (Core0) is supposed to execute the integrity checker with a novel configuration, including appropriate power and hardware configurations (e.g., DTU, hash algorithm IC) to provide reasonable resource consumption for security purposes. In fact, it is not necessary to equip a local memory to a core if no applications in the target OS are using it. Adding an additional monitor core can also help reduce the number of modifications made to the target OS required due to not using Core0 in the original LLM machine architecture.

- A small piece of reserved shared memory space for storing the swap content of the monitor OS. This reserved space can avoid modifications to the target OS to avoid touching this part of the shared memory space. Together with the previous recommendation, this recommendation allows the target OS to run in the LLM machine architecture without any modifications, which substantially reduce the design effort.

- Trusted boot for the secure pager and the monitor OS. We recommend to loading the secure pager and monitor OS images from a secure remote server or an on-chip read-only memory to ensure the integrity of the images. However, if these images need to be loaded from local storage, we prefer to add some hardware units such as the TPM to help boot up the secure pager and the monitor OS to a normal working state.

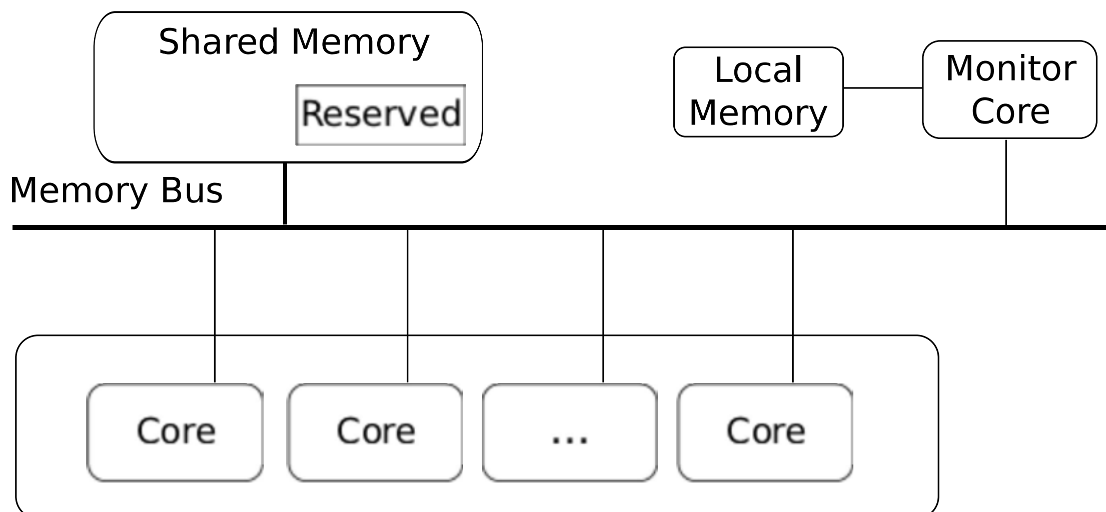

Figure 4.1: Recommended hardware architecture.

5

Conclusion

In this dissertation, we propose an extensible architecture to enhance the security of the monitor OS for improved overall security with low overhead for the target OS.

In the first chapter, we introduce the background and the motivation of our research. We introduce the issue of security in current computing environments and describe why we began our current investigations. We then briefly introduce our own contributions and the structure of this dissertation.

In the second chapter, we focus on the embedded field and introduce the use of embedded hypervisors and microkernels to provide isolation for guest OSs. This topic leads to the introduction of the LLM machine architecture, which can enhance the integrity checker through a hardware-centric method, and thus can provide some characteristics suitable for embedded systems, such as minimal overhead, a small trusted code and minimal modification to the target OS. However, The LLM machine architecture still has some limitations, including that it is only emulated in QEMU, which is a virtual machine monitor, and assumes a limited but rather large local memory that may rarely be equipped on embedded processors.

In the third chapter, we discuss the design, implementation details and evaluation of our system. We first introduce our basic system architecture. Because of the small size of the local memory in RP1, we adapt a software component, the

secure pager, to virtually extend the local memory using a paging mechanism. We also utilize a software-managed TLB to reduce the size of the page tables of the monitor OS, which can reduce the local memory space occupied by the secure pager. Furthermore, we add extensibility to our system by automatically updating the integrity checker. We next discuss the security of our architecture. In the implementation section,we present all of the necessary details regarding the design method. The evaluation section discusses many tests that we performed to demonstrate the security and the performance of our system.

In the fourth chapter, we discuss our research beginning with the introduction of related hardware processor features, including the hardware virtualization extensions, cache architecture and TLB architecture. Then, we discuss whether they can be equipped on future embedded processors based on our analysis. We further discuss our research and propose some hardware recommendations that may generalize the application of the LLM machine architecture.

Our architecture is implemented on a real embedded platform with a hardware configuration similar to LLM but equipped with a much smaller local memory. This implementation provides a robust isolation between the integrity checker and the target OS based on local memory, a hardware feature. To generalize this architecture, we employ a secure pager to virtually extend the physically small local memory space using a swap mechanism with integrity checking for the integrity checker. The secure pager can also update the integrity checker to extend the security functions without disturbing the running of the target OS. Comprehensive evaluations are made within our framework, using one instance of embedded Linux as the target OS and an isolated integrity checker running with the secure pager. The results demonstrate the functions of the secure pager and its influence of the secure pager on Linux in our system. We also present some hardware recommendations to make the LLM architecture more applicable to real embedded processors. Our research illustrates the efficiency of the LLM architecture and generalizes its application to embedded systems by reducing the local memory size. On processors with a suitable architecture, we can build an extensible and secure architecture with reasonable resource consumption, without the issue of heavy overhead interfering the target OS.

6

Future Work

Based on the discussions presented in the previous chapter, we have developed future plans for our continued research.

First the current prototype we built on the RP1 does not include a hypervisor and cannot support multiple target OSs. Our architecture can be extended to support multiple target OSs by inserting a hypervisor, as shown in Fig. 6.1, and the integrity checker can be equipped with more functions to monitor the state of both the hypervisor and the target OSs. This step will further generalize further the application of our research, and we can choose the specific system configuration to suit individual concrete applications.

Next, we should choose the security functions that should be equipped to the integrity checker. It seems that more functionalities usually provide good security. However, in our architecture, more functionalities requires a large integrity checker, and the large size usually requires more page swaps during the loading and execution of the integrity checker, causing heavy overhead. If the execution time of the integrity checker is very long, there will be a security hole between the ending of one security function and the beginning of the next. Therefore, we should select only the necessary functionalities for the integrity checker that are critical for the target OS. However, equipping more functionalities while controlling the size of the integrity checker is another valid approach to achieve the same goal.

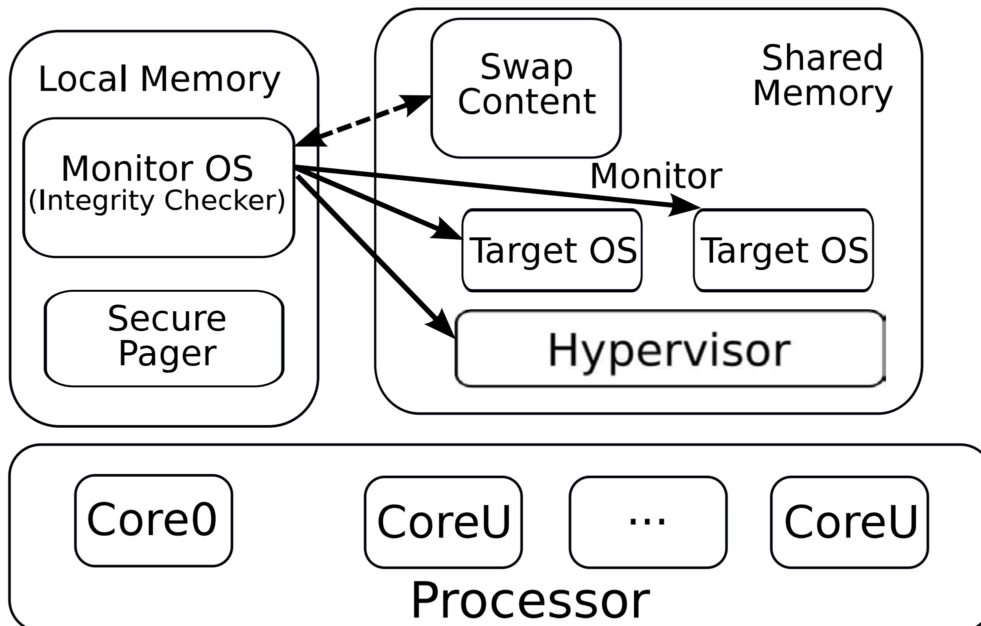

Figure 6.1: Extended system architecture.

Furthermore, more methods should be proposed to optimize the integrity checker. In our research, we have already tried many approaches to construct an integrity checker with good performance. However, to apply this architecture to a real embedded platform, more methods must be evaluated to obtain a reasonable performance. Because different optimizations may be needed for different integrity checkers, a generic method of optimizing the integrity checker would be a valuable approach to solve this issue.

Finally, the software-managed TLB helps reduce the size of the page tables of the monitor OS to efficiently control the size of the secure pager. For processors with hardware-managed TLB, because the format of the page tables is fixed, more effort is required to restrict the size of the page tables, and more kernel modifications to the monitor OS may be needed to reduce the size.

References

[1] **Cortex-A15 Processor**. http://www.arm.com/products/processors/cortex-a/cortexa15.php. 78

[2] **Trusted Platform Module (TPM) based Security on Notebook PCs - White Paper**. http://www.intel.com/design/mobile/platform/downloads/trusted_platform_module_white_paper.pdf, June 2002. 2

[3] **Xv6, a simple Unix-like teaching operating system**. http://pdos.csail.mit.edu/6.828/2012/xv6.html, 2002. 45

[4] **Intel Trusted Execution Technology Architectural Overview**. http://class.ece.iastate.edu/tyagi/cpre681/papers/TXTarch-overview.pdf, 2003. 2

[5] **VMware ESX Server Virtual Infrastructure Node Evaluator's Guide**. http://www.vmware.com/pdf/esx_vin_eval.pdf, 2005. 12

[6] **A Survey on Virtual Machine Security**. http://www.techrepublic.com/resource-library/whitepapers/a-survey-on-virtual-machine-security/, 2007. 4, 12

[7] **Xbox 360 Hypervisor Privilege Escalation Vulnerability**. http://www.securiteam.com/securitynews/5MP040AKUA.html, 2007. 5, 16

[8] **attacking Intel Trusted Execution Technology**. http://invisiblethingslab.com/resources/bh09dc/Attacking2009. 2

[9] **Cloudburst: Hacking 3D (and Breaking Out of VMware)**. http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf, 2009. 5, 16

[10] **An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems.** http://www.eit.lth.se/fileadmin/eit/project/142/virtApproaches.pdf, 2010. 14

[11] **Secured Boot and Measured Boot: Hardening Early Boot Components Against Malware**. http://msdn.microsoft.com/en-us/library/windows/hardware/br259097.aspx, 2012. 2

[12] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. **Comparison of hardware and software cache coherence schemes**. *SIGARCH Comput. Archit. News*, **19**(3):298–308, April 1991. Available from: http://doi.acm.org/10.1145/115953.115982. 81

[13] Thomas J. Ashby, Pedro Diaz, and Marcelo Cintra. **Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters**. *IEEE Trans. Comput.*, **60**(4):472–483, April 2011. Available from: http://dx.doi.org/10.1109/TC.2010.155. 82

[14] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin OfBrien, and Kathryn OfBrien. **A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor**. In Vikram Adve, Mara-Jess Garzarn, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, **5234** of *Lecture Notes in Computer Science*, pages 125–140. Springer Berlin Heidelberg, 2008. Available from: http://dx.doi.org/10.1007/978-3-540-85261-2_9. 81, 82

[15] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. **Automatic Inference and Enforcement of Kernel Data Structure Invariants**. In

Proceedings of the 2008 Annual Computer Security Applications Conference, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society. 17

[16] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. **Scratchpad memory: design alternative for cache on-chip memory in embedded systems**. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM. Available from: http://doi.acm.org/10.1145/774789.774805. 19, 20, 83

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. **Xen and the art of virtualization**. *SIGOPS Oper. Syst. Rev.*, **37**(5):164–177, October 2003. 12, 15

[18] Marti Campoy, A. Perles Ivars, and J. V. Busquets Mataix. **Static Use of Locking Caches in Multitask Preemptive Real-Time Systems**. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, pages 1–6, 2001. 19, 20

[19] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. **Directory-Based Cache Coherence in Large-Scale Multiprocessors**. *Computer*, **23**(6):49–58, June 1990. Available from: http://dx.doi.org/10.1109/2.55500. 81

[20] Chen Chen, JosephB Manzano, Ge Gan, GuangR. Gao, and Vivek Sarkar. **A Study of a Software Cache Implementation of the OpenMP Memory Model for Multicore and Manycore Architectures**. In Pasqua DfAmbra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, **6272** of *Lecture Notes in Computer Science*, pages 341–352. Springer Berlin Heidelberg, 2010. Available from: http://dx.doi.org/10.1007/978-3-642-15291-7_31. 81, 82

[21] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. **Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems**. *SIGARCH Comput. Archit. News*, **36**(1):2–13, March 2008. Available from: http://doi.acm.org/10.1145/1353534.1346284. 12

[22] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. **Breaking up is hard to do: security and functionality in a commodity hypervisor**. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 189–202, New York, NY, USA, 2011. ACM. 5, 15

[23] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. **ReVirt: enabling intrusion analysis through virtual-machine logging and replay**. *SIGOPS Oper. Syst. Rev.*, **36**(SI):211–224, December 2002. 3, 10

[24] Alan M. Dunn, Owen S. Hofmann, Brent Waters, and Emmett Witchel. **Cloaking malware with the trusted platform module**. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 26–26, Berkeley, CA, USA, 2011. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=2028067.2028093. 2

[25] Tal Garfinkel and Mendel Rosenblum. **A Virtual Machine Introspection Based Architecture for Intrusion Detection**. In *NDSS*, 2003. 3, 10

[26] Tal Garfinkel and Mendel Rosenblum. **When virtual is harder than real: security challenges in virtual machine based computing environments**. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, pages 20–25, Berkeley, CA, USA, 2005. USENIX Association. 4, 12

[27] John Goodacre. **Hardware accelerated Virtualization in the ARM Cortex Processors** [online]. Available from: http://www-archive.xenproject.org/files/xensummit_seoul11/nov2/2_XSAsia11_JGoodacre_HW_accelerated_virtualization_in_the_ARM_Cortex_processors.pdf. 78

[28] Google. **Byte-unixbench: A Unix benchmark suite** [online]. Available from: http://code.google.com/p/byte-unixbench/. 53

[29] Brian Hay and Kara Nance. **Forensics examination of volatile system data using virtual introspection**. *SIGOPS Oper. Syst. Rev.*, **42**(3):74–82, April 2008. Available from: http://doi.acm.org/10.1145/1368506.1368517. 3, 10, 11

[30] Joo-Young Hwang, Sang bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. **Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones**. In *Proceeding of 5th IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008. 14, 78

[31] A. Jaleel and B. Jacob. **In-line interrupt handling and lock-up free translation lookaside buffers (TLBs)**. *Computers, IEEE Transactions on*, **55**(5):559–574, 2006. 79

[32] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. **VMM-based hidden process detection and identification using Lycosid**. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, New York, NY, USA, 2008. ACM. 3, 10, 11

[33] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. **NoHype: virtualized cloud infrastructure without the virtualization**. *SIGARCH Comput. Archit. News*, **38**(3):350–361, June 2010. 20

[34] Yuki Kinebuchi, Shakeel Butt, Vinod Ganapathy, Liviu Iftode, and Tatsuo Nakajima. **Monitoring Integrity using Limited Local Memory**. *IEEE Transactions on Information Forensics and Security (TIFS)*, **8**(7):1230–1242, July 2013. 5, 18, 20

[35] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. **Development of Heterogeneous Multi-core Embedded Platform for Automotive Applications**. In *Proceedings of International Conference on Circuits, System and Simulation*, pages 64–75, 2004. 84

[36] Jian Lei, Xia Yang, Guangze Xiong, Wei Jiang, and Yong Liao. **VMM-Based Real-Time Embedded System**. In *Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia*, pages 213–218, Washington, DC, USA, 2008. IEEE Computer Society. 14

[37] Dirk Leinenbach and Thomas Santen. **Verifying the Microsoft Hyper-V Hypervisor with VCC**. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-642-05089-3_51. 12

[38] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. **Why On-chip Cache Coherence is Here to Stay**. *Commun. ACM*, **55**(7):78–89, July 2012. Available from: http://doi.acm.org/10.1145/2209249.2209269. 81

[39] J.M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, V. Gligor, and A. Perrig. **TrustVisor: Efficient TCB Reduction and Attestation**. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158, 2010. 5, 16

[40] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. **How low can you go?: recommendations for hardware-supported minimal TCB code execution**. *SIGARCH Comput. Archit. News*, **36**(1):14–25, March 2008. 2

[41] Shannon Meier. **IBM Systems Virtualization: Servers, Storage, and Software** [online]. Available from: http://www.redbooks.ibm.com/redpapers/pdfs/redp4396.pdf. 9

[42] Jon Millen, Joshua Guttman, John Ramsdell, Justin Sheehy, and Brian Sniffen. **Analysis of a Measured Launch** [online]. Available from: www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA470495? 2

[43] Aashish Mittal, Dushyant Bansal, Sorav Bansal, and Varun Sethi. **Efficient virtualization on embedded power architecture® platforms**. *SIGPLAN Not.*, **48**(4):445–458, March 2013. Available from: http://doi.acm.org/10.1145/2499368.2451163. 14

[44] Sibin Mohan, Jaesik Choi, Man-Ki Yoon, Lui Sha, and Jung-Eun Kim. **SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems**. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 21–32, Washington, DC, USA, 2013. IEEE Computer Society. 20

[45] A. Mousa and A. Hamad. **Evaluation of the RC4 Algorithm for Data Encryption**. *International Journal of Computer Science & Applications*, **3**(2):44–56, 2006. 57

[46] Frank Mueller. **Compiler support for software-based cache partitioning**. *SIGPLAN Not.*, **30**(11):125–133, November 1995. Available from: http://doi.acm.org/10.1145/216633.216677. 81

[47] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. **Improving Xen security through disaggregation**. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM. Available from: http://doi.acm.org/10.1145/1346256.1346278. 12

[48] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, and Tsung-Han Lin. **Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances**. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 645–652, Piscataway, NJ, USA, 2011. IEEE Press. 14, 78

[49] Susanta Nanda and Tzi cker Chiueh. **http://comet.lehman.cuny.edu/cocchi/CMP464/papers/VirtualizationSurveyTR179.pdf** [online]. Available from: http://comet.lehman.cuny.edu/cocchi/CMP464/papers/VirtualizationSurveyTR179.pdf. 9

[50] Tavis Ormandy. **An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments** [online]. Available from: http://taviso.decsystem.org/virtsec.pdf. 12

[51] S. Owicki and A. Agarwal. **Evaluating the performance of software cache coherence**. *SIGARCH Comput. Archit. News*, **17**(2):230–242, April 1989. Available from: http://doi.acm.org/10.1145/68182.68204. 81

[52] 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*, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society. 3, 10, 11

[53] Jinzhan Peng, Guei-Yuan Lueh, Gansha Wu, Xiaogang Gou, and Ryan Rakvic. **A comprehensive study of hardware/software approaches to improve TLB performance for java applications on embedded systems**. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 102–111, New York, NY, USA, 2006. ACM. Available from: http://doi.acm.org/10.1145/1178597.1178614. 79, 80

[54] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. **Copilot - a coprocessor-based kernel runtime integrity monitor**. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, pages 179–194, Berkeley, CA, USA, 2004. USENIX Association. 17

[55] Isabelle Puaut and Christophe Pais. **Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison**. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium. 19, 20, 83

[56] Philip A Reames, Ellick Chan, Francis David, Jeffrey Carlyle, and Roy H. Campbell. **A Hypervisor for Embedded Computing**. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.8391. 14

[57] Daniel Rossier. **EmbeddedXen: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization** [online]. Available from: http://upload.wikimedia.org/wikipedia/commons/5/58/EmbeddedXEN_publication_final.pdf. 14, 78

[58] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. **SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes**. *SIGOPS Oper. Syst. Rev.*, **41**(6):335–350, October 2007. 5

[59] H. Shimada, A. Courbot, Y. Kinebuchi, and T. Nakajima. **A Lightweight Monitoring Service for Multi-core Embedded Systems**. In *Proceeding of 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 202–209, 2010. 39

[60] Kanna Shimizu, Stefan Nusser, Wilfred Plouffe, Vladimir Zbarsky, Masaharu Sakamoto, and Masana Murase. **Cell Broadband Engine™ processor security architecture and digital content**

protection. In *Proceedings of the 4th ACM international workshop on Contents protection and security*, pages 13–18, New York, NY, USA, 2006. ACM. 6

[61] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. **BitVisor: a thin hypervisor for enforcing i/o device security**. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2009. ACM. 5, 12, 15

[62] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. **Efficient computation of sum-products on GPUs through software-managed cache**. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 309–318, New York, NY, USA, 2008. ACM. Available from: http://doi.acm.org/10.1145/1375527.1375572. 81, 82

[63] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. **Reducing TCB complexity for security-sensitive applications: three case studies**. *SIGOPS Oper. Syst. Rev.*, **40**(4):161–174, April 2006. Available from: http://doi.acm.org/10.1145/1218063.1217951. 12

[64] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. **Cache coherence for GPU architectures**. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, **0**:578–590, 2013. 81

[65] James E. Smith and Andrew R. Pleszkun. **Implementation of precise interrupts in pipelined processors**. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98,

pages 291–299, New York, NY, USA, 1998. ACM. Available from: http://doi.acm.org/10.1145/285930.285988. 79, 80

[66] Udo Steinberg and Bernhard Kauer. **NOVA: a microhypervisor-based secure virtualization architecture**. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222, New York, NY, USA, 2010. ACM. 5, 15

[67] I. Tartalja and V. Milutinovic. **Classifying software-based cache coherence solutions**. *Software, IEEE*, **14**(3):90–101, 1997. 81

[68] Allan Tomlinson. **Introduction to the TPM**. In Keith E. Mayes and Konstantinos Markantonakis, editors, *Smart Cards, Tokens, Security and Applications*, pages 155–172. Springer US, 2008. 2

[69] Prashant Varanasi and Gernot Heiser. **Hardware-supported virtualization on ARM**. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 11:1–11:5, New York, NY, USA, 2011. ACM. 14

[70] Xavier Vera, Björn Lisper, and Jingling Xue. **Data cache locking for higher program predictability**. *SIGMETRICS Perform. Eval. Rev.*, **31**(1):272–282, June 2003. 19, 20

[71] D. Wagner and B. Schneier. **Analysis of the SSL 3.0 Protocol**. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, Oakland, California, 1996. 42

[72] Wade Walker and Harvey G. Cragon. **Interrupt Processing in Concurrent Processors**. *Computer*, **28**(6):36–46, June 1995. Available from: http://dx.doi.org/10.1109/2.386984. 79

[73] X. Y. Wang, D. G. Feng, X. J. Lai, and H. B. Yu. **Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD**. Technical report 2004/199, Cryptology ePrint Archive, 2004. 43

[74] T.Y. Wei, Z.L. Qiu, Young C.P., and D.W. Chang. **Single-ISA heterogeneous multi-core architectures for multithreaded workload performance**. In *Proceedings of 31st Annual International Symposium on Computer Architecture*, pages 193–197, 2011. 84

[75] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. **Constructing Services with Interposable Virtual Hardware**. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 169–182, San Francisco, California, USA, 2004. 3, 10, 12

[76] Y. Yoshida, T. Kamei, K. Hayase, S. Shibahara, O. Nishii, T. Hattori, A. Hasegawa, M. Takada, N. Irie, K. Uchiyama, T. Odaka, K. Takada, K. Kimura, and H. Kasahara. **A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption**. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 100–590, 2007. 6, 21, 23

[77] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. **Secure coprocessor-based intrusion detection**. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 239–242, New York, NY, USA, 2002. ACM. 17

Publication List

- **Ning Li**, Yuki Kinebuchi, Hiromasa Shimada and Tatsuo Nakajima. An Extensible Secure OS Architecture for Embedded Systems. *Journal of Information Processing (IPSJ).* Vol. 52, No. 9, September 2010.
- **Ning Li** and Tatsuo Nakajima. Local-Memory-Based Integrity Checking for Embedded Systems. *In Proceedings of IEEE 10th International Conference on Embedded Software and Systems*, December, 2013.
- **Ning Li**, Yuki Kinebuchi, Hitoshi Mitake, Hiromasa Shimada, Tsung-Han Lin and Tatsuo Nakajima. A Light-Weighted Virtualization Layer for Multicore Processor-Based Rich Functional Embedded Systems, *In Proceedings of IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, April 2012.
- **Ning Li**, Yuki Kinebuchi, Tatsuo Nakajima. Enhancing Security of Embedded Linux on a Multi-core Processor, *In Proceedings of IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2011.
- Hitoshi Mitake, Hiromasa Shimada, Tsung-Han Lin, **Ning Li**, Yuki Kinebuchi, Chen-Yi Lee, Daisuke Yamaguchi, Takumi Yajima and Tatsuo Nakajima. Light-Weighted Virtualization Layer for Multicore Processor-Based Embedded Systems. *In Proceedings of International Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, March 2012.
- Hitoshi Mitake, Tsung-Han Lin, Hiromasa Shimada, Yuki Kinebuchi, **Ning Li** and Tatsuo Nakajima. Towards Co-existing of Linux and Real-Time OSes. *In Proceedings of Ottawa Linux Symposium 2011*, June 2011.
- Hiromasa Shinada, Tsung-Han Lin and **Ning Li**. An External Integrity Checker for Increasing Security of Open Source Operating Systems. *Presentation in LinuxCon 2012*, June 2012.