Security System for Connected End-point Devices in a Smart Grid with Commodity Hardware

Hiroshi Isozaki[1, 2], Jun Kanai[1]
[1] Corporate R&D Center, Toshiba Corporation,
Kawasaki, Kanagawa, Japan
{hiroshi.isozaki, jun.kanai}@toshiba.co.jp
[2] Graduate School of Media and Governance,
Keio University,
Fujisawa, Kanagawa, Japan

Shunsuke Sasaki, and Shintarou Sano
Center for Semiconductor Research & Development,
Toshiba Corporation,
Kawasaki, Kanagawa, Japan
{shunsuke.sasaki, shintarou.sano}@toshiba.co.jp

Abstract— **Security has an important bearing on achieving successful commercial deployment of smart grids. In particular, availability is accorded the highest priority in smart grids. For end-point devices, such as smart meters or concentrators, this must be true since they must always be working. We present LiSTEE^{TM} Recovery, an architecture for a fault-tolerant system enabling end-point devices to monitor the status of the operating system and to recover even if they stop working owing to unexpected behavior or cyber-attacks, including zero-day attacks. LiSTEE^{TM} Recovery provides further functions to prevent illegitimate memory modification and to notify a head-end system once a security incident occurs. We demonstrate a full implementation of LiSTEE^{TM} Recovery on a TrustZone-capable ARM-based processor. Our experiment shows that the performance degradation is sufficiently small to be ignored. Furthermore, we observed that the cost of production and maintenance can be minimized.**

Keywords-Smart Grid, Smart Meter, Concentrator, Security, High Availability, TrustZone

I. INTRODUCTION

This paper presents a security system for connected end-point devices in smart grids. It proposes an architecture for a secure fault-tolerant system with commodity hardware and presents a detailed perspective on earlier work by the same authors [1]. In smart grids, requirements for the support of various protocols and functions to network connected end-point devices, such as smart meters or concentrators, make their systems more complicated. Because a large quantity of source code is generally necessary to implement a complicated system, the risk of including vulnerability in the system increases. Moreover, since the devices are connected to home networks, the risk of devices being attacked is high compared with legacy devices connected only to a managed network. In fact, it is reported that smart meters from various vendors were found to improperly handle malformed requests that could be exploited to cause buffer overflow vulnerability; allowing an attacker to cause a system to become unstable or freeze [2]. To keep devices secure in this situation, many security protocols and algorithms have been proposed to securely distribute a shared key between devices and head-end systems or to store privacy data in devices in a secure manner [3][4]. However, confidentiality and integrity are insufficient to solve the security problem in smart grids. Keeping high availability of the devices is strongly desired since they must always be working to provide demand-response services or to use consumption data for payment [5][6]. As a single vulnerability may cause the system to go down, it is very difficult to keep high availability in a complicated system. Furthermore, unlike in the case of interactive devices, such as PCs or smartphones, it is unreasonable to expect end users to reset and restart devices once they freeze or hang since end users cannot recognize the status of the devices and cannot determine whether the device should be rebooted or not. Thus, how to keep the availability of the devices in smart grids is a significant challenge.

To address these problems, we propose LiSTEE^{TM} Recovery, an architecture for fault-tolerant systems that automatically recovers from error status. To achieve this goal, LiSTEE^{TM} Recovery isolates a surveillance process observing the state of the system and a recovery process that reboots the system when it detects the system freezes. In LiSTEE^{TM} Recovery, surveillance and recovery processes run in an isolated secure environment whereas general-purpose processes, including the operating system, such as network or storage access, run in a non-secure environment with hardware access control performed with respect to memory. Hence, a memory area where surveillance and recovery processes are arranged cannot be accessed by general-purpose processes. As a result, even if the operating system is attacked and crashes, it becomes possible to prevent interference in the surveillance and recovery processes.

The remainder of this paper is organized as follows. In Section II, problems are defined. Section III presents background information. Sections IV and V propose a framework and implementation of LiSTEE^{TM} Recovery. The evaluation of LiSTEE^{TM} Recovery is shown in Section VI, related work is referred to in Section VII, and the paper concludes with Section VIII, which is devoted to the conclusion and future work.

II. PROBLEM DEFINITION

In a legacy system, surveillance and recovery processes and their execution environment are monolithically configured. In other words, the reliability of surveillance and recovery processes depends on the reliability of their execution environment. In order to keep reliability high, a

system needs to be implemented without vulnerability. In order to detect and eliminate vulnerability in source code, various testing methods have been proposed [7][8]. However, since end-point devices will be deployed without maintenance over a long period of time within smart grids and new vulnerabilities are found day after day, there is a large risk that such devices will continue operating without vulnerabilities being fixed even if those devices had no vulnerabilities at the time of shipping. For example, there is a well-known attack against x86 processors called "Ret2Libc" which enables an attacker to inject and execute code, and it had been regarded as invalid against ARM processors [9]. However, once new attack which is similar with Ret2Libc against ARM processors has been proposed, buffer over flow on ARM processor has been regarded as real theat. Therefore, attackers may exploit a vulnerability, such as buffer overflow or malformed network input, in order to cause the device to crash. To make matters worse, attackers are in a somewhat advantageous position in launching a large attack since the number of device vendors is limited and the software installed in the devices is uniform. Furthermore, attackers can reverse-engineer code without administrators noticing in order to find a vulnerability since, unlike a server application, devices are located at the user side. Therefore, when attackers find one vulnerability in a single device, they can exploit it on many devices. Considering the above situation, the following problems are to be solved in order to keep high availability under a legacy system.

A. *Difficult to Keep a High Level of Surveillance Continuity*

End-point devices need to support various network protocols and data formats depending on countries or use cases in smart grids [10][11][12]. In order to minimize the implementation cost of a complicated application program or a minor network protocol on end-point devices, Linux will be used as a software execution environment. In Linux, the surveillance and recovery processes can be implemented as a user task executed on the operating system or as an interrupt handler in the operating system. When a surveillance target process is implemented as a user task running on the operating system then support functions in the operating system, such as the "cron" service in Linux, can be used to detect a failure of the user task and to automatically restart the target process. When the surveillance process is implemented as an interrupt handler in the operating system, then more sophisticated implementation is necessary than for an application program; it is automatically and periodically called by a timer interrupt as long as the operating system works. Another legacy approach is implementation of a monitoring and detecting mechanism in the operating system. For example, in order to find buffer overflow attacks, an anomaly detection method is proposed where a protection element monitors system call frequencies, and if the frequencies are different from normal behavior, it determines that an attack occurs [13]. However, the fundamental problem of a legacy approach is that there is no way to restart the process if the operating system itself crashes for any reason. Furthermore, the protection mechanism itself could be a target of the attack, and as a result the protection mechanism could be invalidated. Thus, there is a large risk of devices in a smart grid breaking down and the attack may be able to cause an extensive blackout in the worst case. In order to prevent devices breaking down, a robust method of recovering the system from failure is required in order to keep a high level of availability. Still, some existing hardware devices support a watchdog timer function that detects the status of the operating system and automatically reboots the system [14]. Since not all devices support the function and it is difficult to implement complicated functions in the system as discussed below, a new approach is desired. To clarify the conditions, only a software failure including an attack is assumed in this paper. A physical fault, such as a hardware failure or loss of power, or a hardware attack, such as physically destroying devices or cutting cables, are beyond the scope of this paper.

B. *Difficult for an Administrator to Detect when an Incident Occurs*

End-point devices are connected with a head-end system through the network to provide demand-response services. When the devices detect an error status, such as a surveillance target process being stopped for an unknown reason, it is desirable for these devices to send a report to the head-end system so that an administrator can realize the situation and use the report to investigate the reason for the failure. However, for the reason described above, there is no way for devices to send a message to the head-end system if the operating system crashes in a system where the network connectivity function is implemented as a user task or it is implemented within the operating system. Even in such a case, it is desirable to provide a method enabling devices to send a message to acknowledge the error situation to the system administrator. In addition to the unexpected failure, attacks on the network connectivity function need to be considered. When an attacker gains full access to the system under control, the attacker may try to disable the network connectivity function in the operating system. Therefore, it is desired not simply to provide a method of sending a message but to keep the network connectivity function secure to protect it against the attack even if the operating system is modified or the control of the operating system is taken over.

Besides notification of the error situation to the system administrator, a software update function is also desirable. However, since many existing hardware devices already support a secure firmware update function and its method is highly dependent on each device, it is beyond the scope of this paper.

In addition to the problem described above, the following business problem needs to be considered when introducing a new architecture to the market.

C. *Development and Production Cost*

Cost is an important aspect in evaluating the proposed security architecture. Generally, there are two types of cost: development cost, consisting primarily of personnel expenses, and production cost, which is charged per device.

When implementing an end-point device, if the new security architecture requires a complete software rebuild, the architecture will never be commercialized. Thus, it is desirable to reuse existing software assets, such as libraries, middleware and applications, as much as possible in order to minimize the development cost, including the verification cost. In the case of smart grids, the verification cost is large since reliability is strongly required. Besides the development cost, we need to consider the cost per device. One approach to solve the problems described above is to utilize a dedicated hardware security chip. However, since such chips tend to be very expensive, their use may raise production cost per device. Therefore, the use of widely available existing commodity hardware is desirable in order to minimize production cost.

III. BACKGROUND (TRUSTZONE)

In this section, we provide background information on the hardware technologies leveraged by LiSTEE^{TM} Recovery.

A. ARMv7 Architecture

ARM processors support different processor modes depending on the architecture version. The ARMv7 architecture on which LiSTEE^{TM} Recovery is implemented supports the seven processor modes shown in Table I.

TABLE I. ARM PROCESSOR MODE AND BANK REGISTER

Mode	*level*	*description*	*Bank register*	*# of bank registers*
USR	unprivileged	User mode	r8-r14	7
SVC	privileged	Supervisor mode	r13-r14, spsr	3
IRQ	privileged	IRQ mode	r13-r14, spsr	3
FIQ	privileged	FIQ mode	r8-r14, spsr	8
ABT	privileged	Abort mode	r13-r14, spsr	3
UND	privileged	Undefined mode	r13-r14, spsr	3
MON	privileged	Monitor mode	r13-r14, spsr	3

The processor is executed by selectively switching the modes depending on the process. The processor mode is changed either when a program, such as an operating system, calls a dedicated instruction or when software or hardware exception occurs. The seven modes are categorized as either non-privileged mode or privileged mode by privilege level. In a general system, an operating system is executed in privileged mode and application programs are executed in unprivileged mode. In privileged mode, execution of all instructions and access to all memory regions are allowed, whereas in unprivileged mode availability of instructions and accessibility of memory regions are restricted.

The ARMv7 processor has 40 registers, consisting of 33 general registers and 7 status registers. These registers are arranged in partially overlapping banks. For example, r13, which is a bank register and usually used for stack pointer, refers to different physical registers in User mode and Supervisor mode. For non-banked registers, which refer to the same physical register in different modes, an operating system needs to save and restore in working memory when switching from one mode to another mode so that execution can be subsequently resumed from the same point. On the contrary, the operating system does not need to save the context of banked registers. For example, the operating system does not need to save the context of r13 when switching from User mode to Supervisor mode. Therefore, rapid context switching is enabled.

B. TrustZone

TrustZone is a hardware security function supported by a part of the ARM processor [15][16]. In addition to unprivileged mode and privileged mode, a TrustZone-enabled ARM processor supports two worlds that are independent of the modes. One is the secure world for the security process and the other is the non-secure world for everything else. Each processor mode shown in Table I is available in both the secure world and the non-secure world. Fig. 1 shows the relationship between worlds and modes conceptually. The world in which the processor is executing is indicated by the NS-bit in the Secure Configuration Register (SCR) except when the processor is in monitor mode. When the processor is in monitor mode, it is in the secure world regardless of the value of the NS-bit of SCR. The processor is executed by selectively switching the worlds if necessary. For example, it is assumed that the key calculation process is executed in the secure world and all other general processes, such as storage access or network accesses are executed in the non-secure world.

The software that manages switching between the secure world and the non-secure world is called the monitor. The monitor is executed in monitor mode. TrustZone provides a dedicated instruction, the Secure Monitor Call (SMC) instruction, to transit between the worlds. As soon as the SMC instruction is called, the processor switches to monitor mode. Monitor saves a context of the program running in the current world on the memory and restores a context of the program running in the previous world, then changes the world to set the NS-bit of SCR, and finally executes the program running in the previous world. Besides the SMC

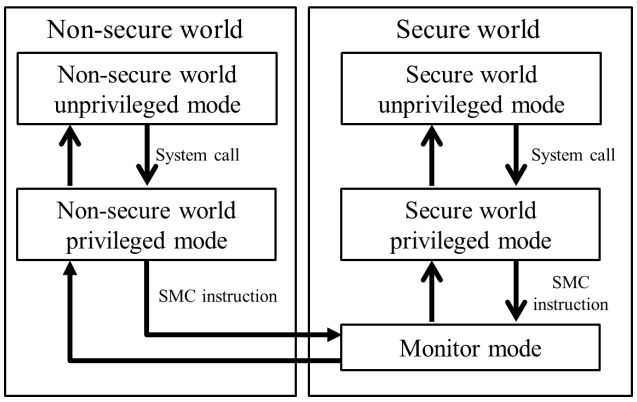

Figure 1. Mode and world in ARM.

instruction, hardware exceptions can be configured to cause the processor to switch to monitor mode.

Note that general registers and Saved Program Status Register (spsr) are not banked between worlds. For example, when r13 in User mode of the secure world is referred and the monitor switches from the secure world to the non-secure world, and then r13 in User mode of the non-secure world is referred, the same physical register is referred. Therefore, the monitor needs to save and restore both bank registers and non-bank registers when it switches worlds.

By using TrustZone-capable hardware, it is possible to make a system where a process running in the secure world can access all system resources, such as memory or peripherals, whereas a process running in the non-secure world can access only a part of system resources. For example, when used in combination with the TrustZone Address Space Controller (TZASC), access to a particular region of working memory can be restricted for a process running in the non-secure world even if the process runs in privileged mode. When a process running in the non-secure world accesses a memory region that it is configured to be prohibited from accessing from a process running in the non-secure world, TZASC generates an interrupt signal and it is sent to the processor. As a result, the violation causes an external asynchronous abort. Similar to TZASC, when used in combination with the TrustZone Protection Controller (TZPC), access to a peripheral can be restricted for a process running in the non-secure world. In contrast to TZASC, the access control policy of TZPC can be configured per peripheral, such as DRAM, Timer, or Real-Time Clock (RTC). That is, the configuration of TZPC is performed peripheral by peripheral. There is a correlation between TZASC and TZPC. For example, when configuring a policy such that access to a particular region of DRAM is restricted, the access control of TZPC corresponding to DRAM is set to off and the proper access control policy with the corresponding region is installed on TZASC. TZPC is configured as secure when booting the system. Therefore, for all peripherals whose access controls are valid by TZPC, access by a process running in the non-secure world is prohibited by default. TZASC and TZPC can only be configured by a process running in the secure world, in order to protect those configurations from illegitimate modification.

IV. Framework of LiSTEE^{TM} Recovery

LiSTEE^{TM} Recovery provides a method for an end-point device to automatically recover from an error status. It also provides a high-level memory protection mechanism. Hence, the recovery process is securely executed without interference. Fig. 2 shows the entire architecture of LiSTEE^{TM} Recovery. LiSTEE^{TM} Recovery consists of three components: Normal OS, LiSTEE^{TM} Tracker Application (LiSTEE^{TM} TA), and LiSTEE^{TM} Monitor.

- Normal OS: An operating system that executes general-purpose processes, such as storage access or network communication. It is executed in the non-secure world. All applications implementing smart meter functions or concentrator functions run on this operating system.
- LiSTEE^{TM} Tracker Application (LiSTEE^{TM} TA): Surveillance and recovery processes executed in privileged mode in the secure world. LiSTEE^{TM} TA includes three modules: Watcher module, Recovery module, and Notification module. The Watcher module is an entry point of LiSTEE^{TM} TA. It is executed periodically by a timer interrupt through LiSTEE^{TM} Monitor. Whenever it is called, it investigates the status of Normal OS. If it detects Normal OS is not working, it calls the Recovery module to reboot the system. Otherwise, it calls the SMC instruction to switch to Normal OS. Moreover, the Notification module is called before the Recovery module reboots the system. It sends a message to notify that the system is about to reboot to the head-end system through network.
- LiSTEE^{TM} Monitor: A program running in the monitor mode. It initializes configurations of TrustZone-related hardware when booting the system. It also provides a context switching function between worlds in the hardware interrupt handler and the SMC handler. Moreover, LiSTEE^{TM} Monitor manages the access control policy and installs the policy on TZASC when booting. Policy Manager takes on their roles.

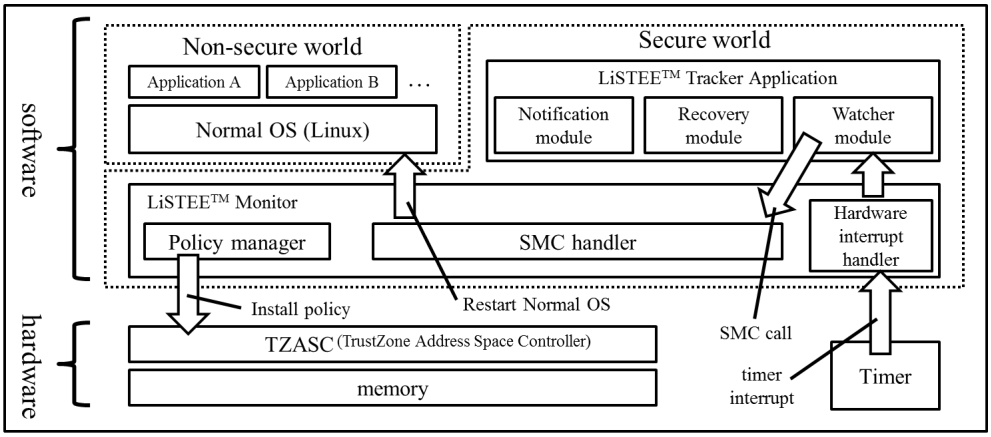

Figure 2. System Architecture of LiSTEE^{TM} Recovery.

The primary feature of LiSTEE^{TM} Recovery is to provide a method for the end-point device to detect the status of Normal OS and to recover it even if Normal OS crashes or stops working. Furthermore, it provides two additional functions. One is to enhance the security protection for LiSTEE^{TM} Monitor, LiSTEE^{TM} TA and Normal OS against attacks. The other is to send a message to the head-end system when an incident occurs. The details of these functions are described below.

A. Baseline Common Functions

LiSTEE^{TM} Monitor has the role of providing baseline common functions to operate Normal OS and LiSTEE^{TM} TA concurrently. LiSTEE^{TM} Monitor has two functions; system initialization and context switching between worlds.

1) System initialization

When booting the system, the processor is in the secure world and LiSTEE^{TM} Monitor is firstly executed. To run Normal OS and LiSTEE^{TM} TA concurrently, it needs to load and execute both of them. It first initializes the status of the processor in both worlds, and loads LiSTEE^{TM} TA in the secure world. Then, it invokes context switching to transit from the secure world to the non-secure world, loads the boot loader program of Normal OS, and executes it in the non-secure world. Finally, the boot loader program loads Normal OS and executes it.

The TrustZone-enabled processor supports the function that is either monitor or Normal OS traps each exception (IRQ, FIQ, and external abort). When booting the system, LiSTEE^{TM} Monitor configures that hardware interrupt handler in LiSTEE^{TM} Monitor traps timer interrupt so that Normal OS cannot interfere with the execution of LiSTEE^{TM} TA when timer interrupt occurs. As well as timer interrupt, LiSTEE^{TM} Monitor configures that hardware interrupt handler in LiSTEE^{TM} Monitor traps external abort. Since the access violation causes external abort as described above, this configuration enables LiSTEE^{TM} TA to detect the occurrence of a memory access violation.

TZPC is configured to be accessed from the secure world only when booting the system. Since Normal OS needs to use peripherals, LiSTEE^{TM} Monitor needs to change the configuration of TZPC to non-secure. The only exception is Timer, which triggers periodical execution of LiSTEE^{TM} TA. Since it is necessary to prevent the configuration of Timer from changing by a process running in the non-secure world, LiSTEE^{TM} Monitor remains the configuration of TZPC corresponding to Timer as secure.

2) Context Switching between Worlds

In LiSTEE^{TM} Recovery, the trigger of context switching between worlds is either the SMC instruction or the Timer interrupt caused by the hardware timer. The SMC handler in LiSTEE^{TM} Monitor is executed when the SMC instruction is called and it transits from the secure world to the non-secure world. In contrast to the SMC handler, the timer interrupt triggers transit from the non-secure world to the secure world. In both cases, LiSTEE^{TM} Monitor invokes context switching between worlds. It first determines the current world. As described in section III-B, general registers and Saved Program Status Register are not banked between worlds. Therefore, LiSTEE^{TM} Monitor needs to save the contents of the registers belonging to the current world on working memory to prevent loss of the previous context, and then change the world. Finally, it restores the contents of the registers belonging to the transition destination world and resumes the execution.

B. Periodical Surveillance and Recovery

While executing Normal OS, whenever the timer interrupt occurs, the processor jumps to the hardware interrupt handler in LiSTEE^{TM} Monitor. The hardware interrupt handler context switches from the non-secure world to the secure world and calls LiSTEE^{TM} TA. Specifically LiSTEE^{TM} Monitor saves a context of Normal OS to memory and restores a context of LiSTEE^{TM} TA, then changes the world and finally calls the Watcher module of LiSTEE^{TM} TA. The Watcher module checks the status of Normal OS. If it judges that Normal OS is not working, the Watcher module calls the Recovery module that reboots the system. Otherwise, it calls the SMC instruction. Then, the SMC handler in the LiSTEE^{TM} Monitor is executed. It context switches from LiSTEE^{TM} TA to Normal OS, and restarts Normal OS at the point just before the timer interrupt occurred. While executing LiSTEE^{TM} Monitor and LiSTEE^{TM} TA, the execution of Normal OS is suspended. That is, Normal OS continues to be processed as if nothing were executed during the execution of LiSTEE^{TM} TA. Fig. 3 shows the flowchart of the periodic surveillance and recovery process.

There are many ways for the Watcher module to determine whether Normal OS is working or not. One of the methods is to check the data area of Normal OS. In general, when an operating system is working, there must be a certain data area that is updated regularly. By checking this data area, it is possible for the Watcher module to judge whether Normal OS is working or not.

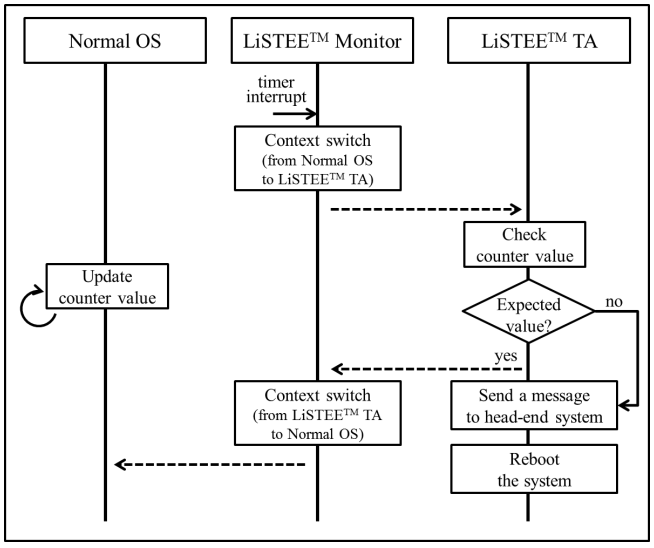

Figure 3. Flowchart of periodical surveillance and recovery.

C. Memory Protection

By utilizing TZASC, LiSTEE^{TM} Monitor provides an access control function such that access of Normal OS running in non-secure mode to the working memory area, which LiSTEE^{TM} Tracker Application running in the secure world uses, is subject to restrictions. Policy Manager in LiSTEE^{TM} Monitor manages three kinds of access control policies: full access, access denied, and read-only. When booting the system, Policy Manager divides working memory into several regions and it installs one of the three access control policies for each working memory region on TZASC before loading Normal OS.

Table II shows how each policy works. Full access indicates no restriction. A process running in both non-secure world and secure world can freely access the region configured according to this policy. This policy is primarily used to share data between Normal OS and LiSTEE^{TM} TA. Access denied indicates full restriction. A process running in the non-secure world can neither read nor write to a region configured according to this policy, whereas a process running in secure world can read and write to the region. Read-only indicates a process running in the non-secure world cannot overwrite the content on the memory, whereas a process running in the secure world can freely access the region using ordinary random access memory, such as DRAM or SRAM, as the working memory which is, of course, physically writable memory.

TABLE II. ACCESS CONTROL POLICY

Policy	*From secure world process*	*From non-secure world process*	
		Read	Write
Full access	OK	OK	OK
Access denied	OK	NG	NG
Read-only	OK	OK	NG

Using these policies, LiSTEE^{TM} Recovery provides two memory protection mechanisms. Fig. 4 shows how these memory protection mechanisms work. One is protection for the kernel area of Normal OS. The other mechanism is protection for LiSTEE^{TM} Monitor and LiSTEE^{TM} TA.

To realize protection for the kernel area of Normal OS, LiSTEE^{TM} Monitor provides read-only memory. In general, when a program is loaded into memory, a data region (data segment) and a code region (code segment) are assigned. In the initial state before booting the system, all regions are allowed to be accessed from the non-secure world by default. In order to allow the boot loader to write the code segment into the memory, LiSTEE^{TM} Monitor leaves the memory region as is until the code segment is loaded. Just after executing the kernel of Normal OS, LiSTEE^{TM} Monitor sets the memory region as read-only for kernel code segment of Normal OS. As a result, even Normal OS is prohibited from overwriting its own code segment.

To protect LiSTEE^{TM} Monitor and LiSTEE^{TM} TA, Policy Manager in LiSTEE^{TM} Monitor installs an access control policy such that Normal OS cannot access the memory area allocated to LiSTEE^{TM} Monitor and LiSTEE^{TM} TA, whereas LiSTEE^{TM} TA and LiSTEE^{TM} Monitor can access all areas when booting the system. This policy protects LiSTEE^{TM} Monitor and LiSTEE^{TM} TA from illegitimate falsification by Normal OS, even if Normal OS is attacked and under the control of an attacker.

Besides the protection for LiSTEE^{TM} Monitor and LiSTEE^{TM} TA, memory protection provides a hardware access control mechanism. One of the possible attacks to disable end-point devices is that of shutting down the system. To prevent such an attack, Policy Manager in LiSTEE^{TM} Monitor installs an access control policy so that Normal OS cannot access the registers corresponding to power management. Thus, it is possible to protect the system against the shutdown attack even if Normal OS is under the control of an attacker.

In the case of policy configured to access denied or read-only, TZASC generates an interrupt signal when the access violation caused by a process running in the non-secure world occurs. LiSTEE^{TM} Monitor configures the hardware interrupt handler in LiSTEE^{TM} Monitor to trap the interrupt so that the system will continues to work without crashing even if access violation occurs, and LiSTEE^{TM} Monitor can detect the access violation.

D. Message Notification

LiSTEE^{TM} Recovery provides a function to notify the head-end system that Normal OS has stopped working and is rebooting the system by sending a message through the

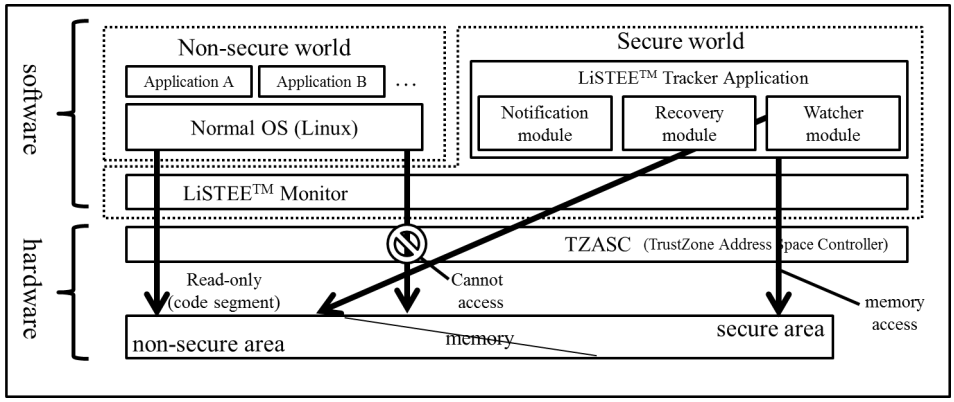

Figure 4. Memory protection mechanism.

network even if the operating system is modified or the control of the operating system is taken over; the resulting network function is disabled by an attacker. The Notification module has the role of sending a message. Although Normal OS has a network connectivity function, such as TCP/IP stack, LiSTEE^{TM} TA cannot use the function since there is a case where it is not working when sending a message. Thus, LiSTEE^{TM} TA supports the network connectivity function including the network application, the network protocol stack and the network driver to notify the error situation to the system administrator through the network. Obviously, it is possible to send a head-end system a message whenever LiSTEE^{TM} TA is executed to notify that the system works correctly.

V. PROTOTYPE IMPLEMENTATION

We used ARM C/C++ Compiler 5.01 to build LiSTEE^{TM} Monitor and LiSTEE^{TM} TA. We used gcc 4.4.1 to build Linux 3.6.1 as Normal OS. We chose Motherboard Express uATX with the CoreTile Express A9x4 processor that supports TrustZone as an execution environment.

Regarding a memory map, from 0x48000000 through 0x4A000000 is assigned for SRAM, and from 0x60000000 through 0xE0000000 is assigned for DRAM. Table III shows the memory map with the access control policy of the memory. In Table III, Normal OS (code) indicates the Linux kernel code. Normal OS (data) includes the Linux data, the application code and the application data. For clarification, full access is applied from the non-secure world for an area not described in Table III.

TABLE III. MEMORY MAP

Data	Start Address	Size	Security Permission (From non-secure world)
Vector tables + Initialization code + LiSTEE^{TM} Monitor + LiSTEE^{TM} TA	0x48000000	0x01B00000	Access denied
Normal OS (code)	0x60000000	0x002FE000	Read-Only
Normal OS (data)	0x602FE000	0x3EF02000	Full access
Shared memory	0x9F200000	0x00C00000	Full access

For the Policy Manager in LiSTEE^{TM} Monitor to install an access control policy on TZASC, the start address and the size of each memory region are predefined. After the boot loader loads Linux at the predefined value, LiSTEE^{TM} Monitor installs the access control policy on TZASC. As shown in Table III, the access to the memory regions allocated to LiSTEE^{TM} Monitor, LiSTEE^{TM} TA and the code segment of Normal OS is restricted for the Normal OS running in the non-secure world, whereas the access to the region allocated to the data segment of Normal OS and shared memory is not. For clarification, LiSTEE^{TM} Monitor and LiSTEE^{TM} TA running in the secure world can access all regions. Furthermore, since LiSTEE^{TM} Monitor sets the configuration registers of TZASC to prohibit Normal OS from accessing them, Normal OS cannot change this configuration.

Table IV shows the configuration of TZASC. In Table IV, the meaning of the value of the security permissions field is as follows: 0b1111 indicates full access from both the secure world and the non-secure world, 0b1100 indicates secure read/write is permitted but non-secure read/write is restricted (access denied), and 0b1110 indicates secure read/write and non-secure read are permitted but non-secure write is restricted (read-only). An entry with larger entry number is accorded higher priority than one with smaller entry number. Therefore, we first set all regions with a policy of full access as entry number 0, and then set access control policies from entry number 1 through 7. The size of a region to which access control is applied is discrete, such as 32 KB, 64 KB, …, 1 MB, 2 MB, 4 MB, …, 2 GB, 4 GB. Therefore, to set policy for LiSTEE^{TM} Monitor and LiSTEE^{TM} TA whose size is 0x01B00000 (27 MB), we used four entries: entry number 1 (16 MB), entry number 2 (8 MB), entry number 3 (2 MB), and entry number 4 (1 MB). In contrast to the size of LiSTEE^{TM} Monitor and LiSTEE^{TM} TA, the size of Normal OS (code) is a fraction (32 MB – 8 KB), and TZASC has restrictions such that it is impossible to define an entry whose size is smaller than 32 KB. Instead, it is possible to define a subregion to equally divide a region into eight with the access control policy, and enable the policy for each subregion. For example, when the size of a region is 32 KB, it is possible to enable a policy for each 4 KB subregion. An 8 bit subregion disable field controls enabling and disabling the policy. Each bit in a subregion disable field enables the corresponding subregion to be disabled. For example, when zero is set to the value of the highest bit in a subregion disable field, the policy for subregion 0 (the subregion having the highest address) is enabled. To set the policy for a Normal OS (code) region, we first defined two regions, 2 MB (entry number 5) and 1 MB (entry number 6) and set the read-only policy. Then, we defined the region with a size of 64 KB (entry number 7) that overlaps the last portion of entry number 6, equally divides the region into eight, sets the policy of full access, and enables the policy for the last subregion only. As a result, the policy of full access is set to the subregion having the highest address only, and the policy of read-only remains for the rest of the subregions.

As shown in Table III and Table VI, the policies can be clearly defined and there is no overlapped region. Thus, no policy conflict exists in LiSTEE^{TM} Recovery.

TABLE IV. CONFIGURATION OF TZASC

Entry Number	Start Address	Size	Subregion disable	Security Permission
0	--	--	--	0b1111
1	0x48000000	0x17(16MB)	0x0	0b1100
2	0x49000000	0x16(8MB)	0x0	0b1100
3	0x49800000	0x14(2MB)	0x0	0b1100
4	0x49A00000	0x13(1MB)	0x0	0b1100
5	0x60000000	0x14(2MB)	0x0	0b1110
6	0x60200000	0x13(1MB)	0x0	0b1110
7	0x602F0000	0xF(64KB)	0x7F	0b1111

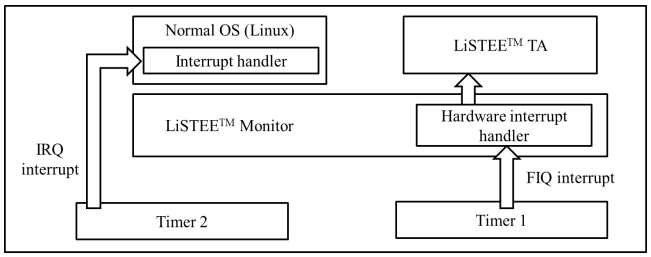

Figure 5. Assignment of timer interrupt.

Fig. 5 shows the assignment of the timer interrupt. We allocated a timer interrupt caused by a timer (timer 1) to Fast Interrupt Request (FIQ) and the timer interval was set to 1 second. The FIQ interrupt is handled by the hardware interrupt handler in LiSTEE™ Monitor, then it calls LiSTEE™ TA and, as a result, LiSTEE™ TA is periodically called. We used another timer (timer 2) and allocated it to Interrupt Request (IRQ), and the timer interval was set to 4 milliseconds. The IRQ interrupt is handled by the interrupt handler in Linux. Since Linux assumes the timer interrupt is allocated to IRQ, modification of the Linux source code to adopt LiSTEE™ Monitor is unnecessary.

Table V shows a configuration of hardware interrupt. We configured Secure Configuration Register (SCR) and Current Program Status Register (CPSR) so that the FIQ handler of LiSTEE™ Monitor is called when the FIQ interrupt occurs, whereas the IRQ handler in Linux is called when the IRQ interrupt occurs during executing Linux. Table VI shows the register setting to achieve the configuration of Table V. CPSR.I indicates the Interrupt disable bit and is used to mask the IRQ interrupt. CPSR.F indicates the Fast interrupt disable bit and is used to mask the FIQ interrupt. CPSR.A indicates the asynchronous abort disable bit and is used to mask asynchronous abort. SCR.FIQ controls which mode the processor enters when the FIQ interrupt occurs. If one is set, it enters monitor mode, otherwise it enters FIQ mode. SCR.IRQ controls which mode the processor enters when the IRQ interrupt occurs. If one is set, it enters monitor mode, otherwise it enters IRQ mode. SCR.FW controls whether the F bit in the CPSR can be modified in the non-secure world. SCR.EA controls which mode the processor enters when external abort including the one generated by TZASC. If one is set, it enters monitor mode, otherwise it enters abort mode. SCR.AW controls whether the A bit in the CPSR can be modified in the non-secure world. If zero is set, CPSR.A can be modified only in the secure world, otherwise it can be modified in both worlds.

TABLE V. RELATIONSHIP BETWEEN WORLD AND INTERRUPT

World when interrupt occurs	Interrupt	Jumps to
Non-secure world	FIQ	Hardware interrupt handler (FIQ handler) in LiSTEE™ Monitor
	IRQ	IRQ handler in Normal OS (Linux)
Secure world	FIQ	Pending FIQ
	IRQ	Pending IRQ

TABLE VI. CPSR AND SCR REGISTER SETTING

		Non-secure world	Secure world (LiSTEE™ TA)	Secure world (Monitor)
CPSR	I	0/1 (depending on the configuration of Normal OS)	1 (IRQ disabled)	1 (IRQ disabled)
	F	0 (FIQ enabled)	1 (FIQ disabled)	1 (FIQ disabled)
	A	0 (Asynchronous abort enabled)	0 (Asynchronous abort enabled)	1 (Asynchronous abort disabled)
SCR	FIQ	1 (enter monitor mode)	0 (enter FIQ mode)	0/1 (depending on which world transiting to)
	IRQ	0 (enter IRQ mode)	0 (enter IRQ mode)	0 (enter IRQ mode)
	FW	0 (can be modified CPSR.F only in secure)	0 (can be modified CPSR.F only in secure)	0 (can be modified CPRS.F only in secure)
	EA	1 (enter monitor mode)	0 (enter abort mode)	0/1 (depending on which world transiting to)
	AW	0 (can be modified CPSR.A only in secure)	0 (can be modified CPSR.A only in secure)	0 (can be modified CPSR.A only in secure)

As shown in Table V, when a processor is in the non-secure world and the FIQ interrupt assigned for timer 1 occurs, the FIQ handler in monitor mode is called since one is set to SCR.FIQ. The FIQ handler in monitor mode switches from the non-secure world to the secure world and calls the FIQ handler in LiSTEE™ TA. Finally, the FIQ handler in LiSTEE™ TA calls the Watcher module. The entry point to LiSTEE™ TA from LiSTEE™ Monitor is only the FIQ handler in LiSTEE™ TA and it never returns to LiSTEE™ TA after the Watcher module calls SMC instruction under the current implementation. When considering returning to the original location in LiSTEE™ TA when entering the secure world next time as future extension, the FIQ handler in monitor mode sets the instruction located in the address next to the address of the instruction just after calling the SMC instruction in the previous time to r14 before calling the FIQ handler of LiSTEE™ TA. On the other hand, when the IRQ interrupt occurs, the IRQ handler in Normal OS is called. Furthermore, Normal OS cannot change the configuration of CPSR.F since zero is set to SCR.FW. Therefore, the FIQ interrupt is always enabled and the timer interrupt is input to the monitor.

When a processor is in the secure world and FIQ or IRQ interrupt occurs, the interrupt is pending since zero is set to CPSR.F and CPSR.I. For future extension, LiSTEE™ Monitor changes SCR.FIQ setting during context switching so that LiSTEE™ TA handles the FIQ interrupt directly without LiSTEE™ Monitor when the FIQ interrupt occurs in the secure world. That is, zero is set to SCR.FIQ when it transits from the non-secure world to the secure world to jump to the FIQ handler in LiSTEE™ TA when the FIQ interrupt occurs in the secure world. On the other hand, one is set when it transits from the secure world to the non-secure world to enter monitor mode when the FIQ interrupt occurs in the non-secure world.

When a processor is in monitor mode, FIQ and IRQ interrupt are disabled to avoid occurrence of multiple interrupt.

In order to determine whether Linux is working or not, we made a small application program, which runs on Linux and communicates with LiSTEE^{TM} TA. Shared memory is used to exchange data between LiSTEE^{TM} TA and Normal OS. The application program writes a counter value into the shared memory periodically. Then LiSTEE^{TM} TA reads the counter value from the shared memory. When Normal OS is crashed, the application program cannot update the counter value. If the counter value is not updated in a certain amount of time or the counter value is not an expected value, LiSTEE^{TM} TA determines that Normal OS is not working. Another method of checking the status of Normal OS is to monitor the status of a specific field, such as a task structure or page tables in Normal OS, but we have not implemented it. Thanks to the memory protection function, it is impossible for Normal OS to check the checking process running in LiSTEE^{TM} TA. Since it is possible to maintain secrecy of Normal OS as to which memory area of Normal OS LiSTEE^{TM} TA monitors or how often LiSTEE^{TM} TA checks it, it is difficult for an attacker to plan a countermeasure to circumvent the checking.

LiSTEE^{TM} Recovery provides a method to continue working even if a memory access violation caused by TZASC occurs. Fig. 6 shows the flowchart of how LiSTEE^{TM} TA and LiSTEE^{TM} Monitor recover from the error status to the normal status when an access violation caused by TZASC occurs. When booting the system, LiSTEE^{TM} Monitor configures SCR.EA so that external aborts including the ones TZASC generates are handled in Monitor mode, instead of by the abort handler in Normal OS. Furthermore, it is prohibited to mask external abort from the non-secure world to configure SCR.AW. Therefore, when an access violation occurs in user mode in the non-secure world, for example, a processor jumps to the abort handler in LiSTEE^{TM} Monitor. At this time, the values of r14 (lr) and spsr are the values of PC (Program Counter) and spsr of the mode just before the access violation occurs, respectively. The abort handler in LiSTEE^{TM} Monitor saves registers including r14 and spsr of original mode in the non-secure world on working memory, context switches from the non-secure world to secure world, and calls the abort handler in LiSTEE^{TM} TA. The abort handler in LiSTEE^{TM} TA checks the status of Normal OS. For example, LiSTEE^{TM} TA checks which process running in Normal OS triggers access violation or checks memory address where an access violation is triggered to investigate the reason for the access violation later. After LiSTEE^{TM} TA checks the status, it calls the SMC instruction and jumps to LiSTEE^{TM} Monitor. While LiSTEE^{TM} TA works in the background when an access violation occurs, LiSTEE^{TM} Recovery behaves as if data abort occurs from the viewpoint of Normal OS. When data abort occurs, a processor automatically stores PC and cpsr of the mode just before data abort occurs to r14 and spsr respectively. LiSTEE^{TM} Monitor carries out a similar operation with the processor when an access violation occurs. LiSTEE^{TM} Monitor switches from the secure world to the non-secure world, restores the saved values including setting the saved value of r14 and spsr just before the access violation occurs to banked registers for abort mode in order to be able to return to the original location after exiting abort mode, and calls the abort handler of Normal OS. Therefore, when Normal OS restarts a process, the data abort handler is executed.

When LiSTEE^{TM} TA determines that Linux is not working, it sends the head-end system a message. In order to send a message to the head-end system when LiSTEE^{TM} TA detects that Linux is not working, we ported a network driver and UDP/IP stack to LiSTEE^{TM} TA. We defined a proprietary protocol and data format over UDP to notify the head-end system that LiSTEE^{TM} TA starts reboot of the system. An application data size of UDP packet is 32 bytes, and it consists of 4 bytes of device ID, 1 byte of flag indicating the status of the device, and 27 bytes of reserved area.

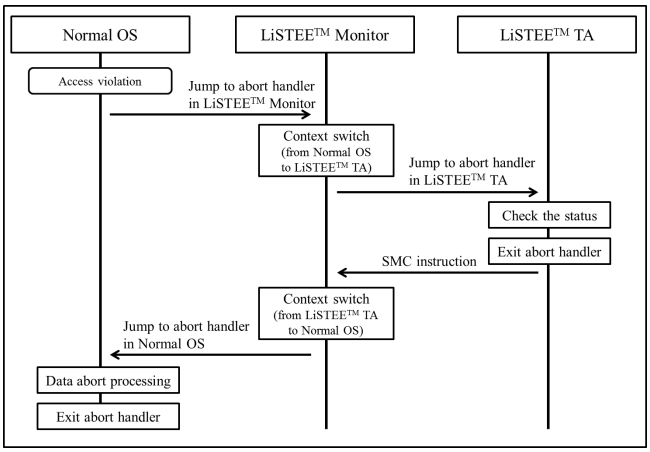

Figure 6. Flowchart of access violation handling.

VI. EVALUATION

In this section, we describe the result of the evaluation in terms of security to verify the problems of the legacy system defined in Section II can be solved. Performance and cost analysis of LiSTEE^{TM} Recovery is also described below.

A. Security Analysys

1) Surveillance and Recovery: LiSTEE^{TM} Recovery can recover from a failure to reboot the system even if Normal OS crashes. The reason for the crash could be a software bug or a cyber-attack, including a zero-day attack prompted by unknown vulnerabilities. In either case, since the hardware timer interrupt continues working regardless of the state of Normal OS, LiSTEE^{TM} TA is always periodically called and can detect a failure of Normal OS. At the next level, it is desirable to detect the failure as soon as possible. Detection time depends on how frequently LiSTEE^{TM} TA checks the status of Normal OS. Since the execution time of LiSTEE^{TM} TA and context switching by LiSTEE^{TM} Monitor is very short, LiSTEE^{TM} Recovery can detect the crash of Normal OS very quickly. Some attackers may continue to

attack just after rebooting the system. One possible approach to a countermeasure for the attack is to let LiSTEE^{TM} TA have a minimum function like the "safe mode", but we have not implemented that.

2) Attack Prevention: The proposed system provides two levels of attack prevention mechanism. The first level is to prevent Normal OS from illegitimate modification. When an attacker gains full control of Normal OS to misuse the vulnerability, the attacker may overwrite the code segment of Normal OS to directly overwrite the memory. In fact, many vulnerabilities (e.g., CVE-2013-4342, CVE-2013-1969, and CVE-2008-1673) allowing a remote attacker to execute arbitrary code are reported [17]. In the case of Linux, for example, once arbitrary code is executed with an administrator privilege by an attacker, it is possible for the attacker to overwrite an arbitrary area of code segment through /dev/mem, resulting in system crash or misbehavior. Although overwriting the code segment in memory is generally difficult, it is relatively easy in the case of end-point devices since the hardware configuration is fixed. As a result, the system may go down. However, since LiSTEE^{TM} Monitor sets the access control of the memory region for the code segment of Normal OS as read-only, and its configuration can be changed only from the secure world, it is impossible for Normal OS to overwrite the code segment of Normal OS. An advantage is that the protection does not cause any side effects. Since a data segment is used to store the state of the program, Normal OS updates the content of the data segment frequently during its execution. In contrast to the data segment, since a code segment is used to store program code, it is not expected to update its content after booting the system. In particular because devices such as smart meters or concentrators are not expected to change their function after being deployed, the dynamic update function is not required. Thus, this protection mechanism can protect Normal OS from illegitimate modification without side effects. Moreover, the feature of read-only memory is vey useful for the data, whose value is only changed by LiSTEE^{TM} TA and to which Normal OS only refers. The typical application is a secure clock. In a legacy system, it is very difficult to provide a secure clock on an operating system without network connectivity or dedicated hardware if illegitimate modification of the operating system is premised. However, LiSTEE^{TM} TA can provide a local secure clock function by software. Since LiSTEE^{TM} TA is executed periodically and it knows the frequency of the execution, it is possible for LiSTEE^{TM} TA to update a counter value written in a read-only memory in a certain amout of time periodically. Because the counter value is read-only from Normal OS, Normal OS cannot revert the counter value. The second level is to protect LiSTEE^{TM} Monitor and LiSTEE^{TM} TA from illegitimate modification and suspension. Since the first level of protection is effective only for a code segment of Normal OS, an attack that overwrites a data segement cannot be prevented. Thus, there are still possibilities that control of Normal OS is gained by an attacker. Even in such cases, thanks to TZASC, since Normal OS is prohibited from overwriting the content of memory where LiSTEE^{TM} TA and LiSTEE^{TM} Monitor are allocated, illegitimate modification is prevented. Since communication interface between Normal OS and LiSTEE^{TM} TA is limited, it is impossible to compromise LiSTEE^{TM} TA by an attack. Moreover, since the interrupt configuration register is accessible only from the secure world, there is no way for Normal OS to stop the timer interrupt. Furthermore, LiSTEE^{TM} provides a mechanism to protect against shutdown attack. Since it is impossible to prevent Normal OS from executing a shutdown procedure with a priviledged instruction in the non-secure world, when a process running in the non-secure world tries to shutdown the system, LiSTEE^{TM} TA can detect it and discard the shutdown request. Since end-point devices usually keep working all the time, devices could be implemeted without having a shutdown or reboot function. However, it is necessary to have a shutdown function in some cases. For example, the system may need to reboot when updating firmware. Another example is that a service enginner may need to reboot the system when inspecting the status of the end-point devices for maintanance purposes. Although it has not been implemented, it is possible to endow LiSTEE^{TM} TA with a function to determine whether it should shutdown or not based on the status of the system. For example, when LiSTEE^{TM} TA detects an access to the memory region mapped to the registers corresponding to power management and determines that the system is under a particular status, such as a maintanance mode, it may allow executing a shutdown procedure. Similarly, when LiSTEE^{TM} TA detects the access, it sends a head-end system a message to inquire whether the shutdown request is accepted or not by using the message notification function. Based on a response to the inquiry, it can determine whether or not a shutdown procedure can be executed without interference of Normal OS.

3) System Reliability: In a legacy system, one single bug could affect the entire system, causing a critical failiure. Ideally, from a defensive viewpoint, the entire system including the operating system should be bug-free to achieve high availability. However, it is impracticable to build a complicated system without bugs. Linux 3.6.1 consists of over 15 million lines of code and many new bugs that cause critical crash are reported frequently (e.g., CVE-2013-4563, CVE-2013-4387, and CVE-2012-2127) even though it is carefully reviewed by many professionals [17]. Thus, the smaller the critical component that has to be robust within a system, the better. In the case of LiSTEE^{TM} Recovery, the critical components correspond to LiSTEE^{TM} TA and LiSTEE^{TM} Monitor. In contrast to Linux, the code size of LiSTEE^{TM} Monitor and LiSTEE^{TM} TA is relatively

small. The volume of source code for LiSTEE^{TM} Monitor is about 700 lines and its code and data size are 2.1 KB and 1.6 KB, respectively. Similarly, the volume of source code of LiSTEE^{TM} TA is about 41200 lines and its code and data size are 1.09 MB. Compared to the volume of source code of Linux, the risk of LiSTEE^{TM} Monitor and LiSTEE^{TM} TA including bugs is small.

4) Response to Failure: The Notification module in LiSTEE^{TM} TA sends a message to the head-end system just before rebooting the system. The message, which notifies that particular devices are about to reboot, is sometimes useful information for administrators. For example, if messages are sent by devices having a particular software version number, the reboot could be caused by an attack aimed at a vulnerability specific to the software. If messages are sent by devices located in one particular network, the reboot could be caused by a network worm distributed in that specific network. Although LiSTEE^{TM} Recovery cannot prevent an attack in advance, the notification feature can help the administrator investigate the reason for the failure during or after the incident. For example, it is impossible for LiSTEE^{TM} Recovery to prevent an attacker from compromising Normal OS and causing reboot frequently. However, the administrator can notice that frequent reboot occurs to the device through network since Notification module sends a message each time when rebooting. The attackers may try to block sending of the message to circumvent the notification. However, Normal OS cannot interfere with the Notification module sending a message to the head-end server since the Notification module is executed inside LiSTEE^{TM} TA. Moreover, since LiSTEE^{TM} TA is processed in an environment isolated from Normal OS, security processes, such as encrypting a message, are easy to implement in LiSTEE^{TM} TA. Therefore, once an encryption key and an encryption process are implemented in LiSTEE^{TM}, it is possible to keep them secret from Normal OS. In the next step, it is possible to include a firmware update feature to implement functions receiving data from the head-end system and writing the data into the file system to extend the function of the Notification module. In combination with the "safe mode" described above, this function is effective against a continuous attack that occurs just after the system recovers.

B. Performance Analysis

As well as the implementation environment, we used Motherboard Express uATX that contains the ARM Cortex-A9x4 processor running at 400 MHz as an experimental environment. Level 1 instruction cache, level 1 data cache, and level 2 cache are 32 KB, 32 KB, and 512 KB, respectively. It contains 1 GB DRAM as the main memory and we assigned the same memory map as that described in Section V.

First, we measured the execution time of LiSTEE^{TM} TA during execution of Normal OS; to be precise, the time period from the beginning of the hardware interrupt handler in LiSTEE^{TM} Monitor through to the execution of the SMC instruction. Without calling the Notification module, the average time is 1.7 microseconds over 10,000 trials. However, if the Notification module is called, the average time is 4.1 milliseconds over 10,000 trials. Note that the Notification module is called when rebooting the system, which rarely occurs. Thus, this performance overhead poses no problem.

Next, we measured the performance degradation of Normal OS. Since the execution of Normal OS is suspended during execution of LiSTEE^{TM} TA, the performance of Normal OS degrades in any case. The total of Normal OS suspension time depends on the frequency of calling LiSTEE^{TM} TA. There is a tradeoff between the performance degradation of Normal OS and the delay in detecting the crash of Normal OS. When the frequency is increased, the performance degradation of Normal OS is also increased. On the other hand, when the frequency is decreased, the delay for detecting the crash of Normal OS becomes larger. Since a general application is assumed to be executed on Normal OS, we used dhrystone as a benchmark program to measure the performance degradation [18].

Fig. 7 shows the result of the experiment. The bar graph

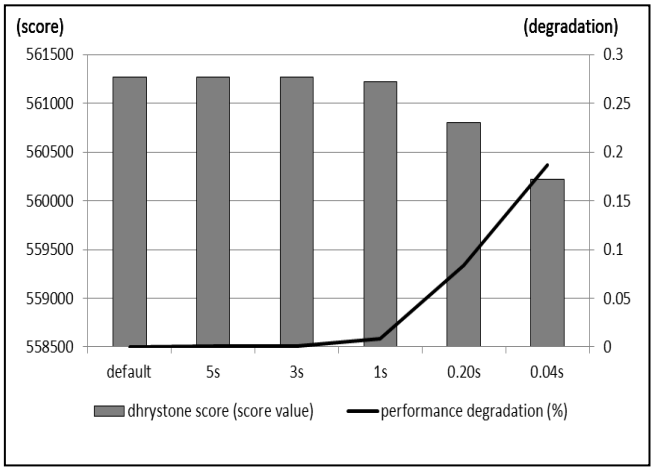

Figure 7. Result of performance degradation.

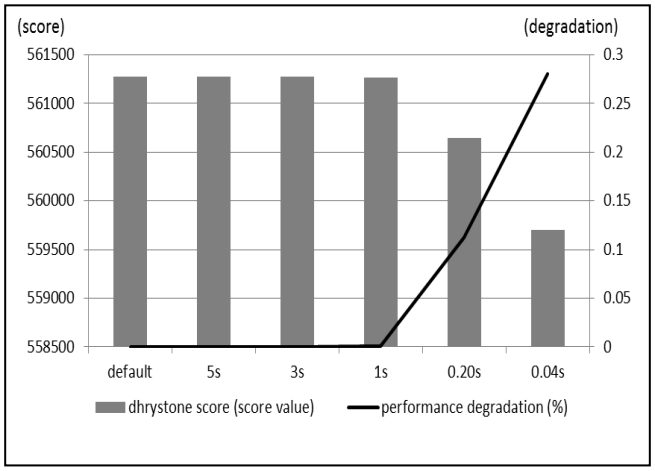

Figure 8. Result of performance degradation with message transmission.

shows the dhrystone score and the line graph shows the performance degradation. The higher the score, the better the performance is. Each bar shows the timer interval of calling LiSTEE^{TM} TA and its value is default (never called), 5 seconds, 3 seconds, 1 second, 0.2 seconds and 0.04 seconds respectively. When the timer interval was set to 5 seconds, the performance degradation was suppressed within 0.001 %. Even if the interval was set to 0.04 seconds, the performance degradation was less than 0.2 %. The result shows that although there is a tradeoff between performance degradation of Normal OS and detection rate logically, the performance degradation can be ignored in practice even if the frequency of calling LiSTEE^{TM} TA is increased. Fig. 8 shows another result of the experiment. In the case of Fig. 7, it is assumed that the Notification module sends a head-end system a message only when Normal OS stops working and the system is rebooting. Therefore, the result does not include processing time of the Notification module. On the other hand, Fig. 8 assumes that the Notification module sends a head-end system a 32 byte message whenever LiSTEE^{TM} TA is executed even if Normal OS is working correctly. This experiment assumes that Notification module sends the head-end system a message periodically even if Normal OS keeps working so that an administrator can monitor the status of each device. Although the result of the experiment shows that the performance slightly degrades compared with the experiment without message transmission, it can still be ignored in practice. Note that the score was better for the experiment with message transmission than for the experiment without message transmission when the interval was set to 5 seconds, 3 seconds and 1 second. When the timer interval is long, the execution times of LiSTEE^{TM} TA and LiSTEE^{TM} Monitor are negligible compared with the execution time of Normal OS since the task is too small to measure accurately. Thus, this can be regarded as an error.

C. Cost Analysis

1) Development Cost: LiSTEE^{TM} Recovery does not require any modification to Linux in order to run it as Normal OS on LiSTEE^{TM} Monitor. Thus, in terms of application developer's cost, since developers can reuse all existing programs including libraries, middleware, and applications running on Linux, no additional develpment cost is necessary. In terms of device developper's cost, configuration, such as network address setting of Notification module, and memory address setting and security permission setting of TZASC is necessary to integrate LiSTEE^{TM} Recovery into a device. In addition to the development cost, verification cost in order to check that the configuration is correct is necessary. For embedded devices in Smart Grid, there are cases where the performance requirement is specified. For example, in the case of a smart meter, it is reported that an acceptable delay in responding to a management server is in the range of 50 ms to 300 ms under a specific condition [19]. As described in the performance analysis, since performance degradation is insignificant when introducing our proposed method, the cases requiring performance tuning are limited. Therefore, the development cost can be controlled.

2) Production Cost: LiSTEE^{TM} Recovery is software-based technology and no additional hardware except a TrustZone-capable ARM processor and an address space controller is required. TrustZone-capable processors are widely available. In fact, all ARM Cortex A series processors support TrustZone. Therefore, the additional cost is mitigated. As a result, development cost per device can be minimized.

3) Maintenance Cost: It is assumed that a tremendous number of devices will be deployed in the field for smart grids. In the case of a cyber-attack, since many devices could be a target of the attack and the attack could be done in a very short period of time through the network, it is impracticable in terms of both cost and time for field service engineers to physically visit each site and reboot them. The autorecovery feature of LiSTEE^{TM} Recovery mitigates this problem. Moreover, the report is sent to the head-end system once the device reboots. This function contributes to reduction of the cost of troubleshooting. Thus, LiSTEE^{TM} Recovery provides an opportunity to reduce maintainance cost compared with legacy systems.

VII. Related work

To recover from an operating system failure, various approaches have been proposed.

The simplest approach is that of including the recovery mechanism within the operating system. One method is to use Non-maskable Interrupt (NMI) as a watchdog timer [20]. NMI is a processor interrupt that cannot be ignored. When NMI is generated, the NMI handler implemented within the operating system is called regardless of the status of the operating system. Since it is not necessary to save and restore registers to execute a process implemented in NMI handler, performance overhead is mitigated. Thus, NMI can be used as a surveillance and recovery process to implement the NMI handler so that it detects whether the operating system hangs or not. In [21], Dolev et al. propose self-stabilizing operating system by utilizing NMI. Although NMI is easy to use as a watchdog timer because it has already been implemented in Linux, it is vulnerable because the NMI handler could be invalidated to overwrite the code segment of the operating system. Furthermore, since implementation of a rich application in an interrupt handler, such as a network communication function or a data encryption function, is not anticipated, it is difficult to realize the notification function.

Another approach to recover from the failure is to check the status of the operating system from outside using virtualization technology. It is easy to realize an isolation environment by utilizing virtualization technology. Karfinkel developed the trusted virtual machine monitor (TVMM), on which a general-purpose platform and a special-purpose platform executing security-sensitive processes run separately and concurrently [22]. The libvirt project develops a virtualization abstraction layer including a virtual hardware watchdog device [23]. To cooperate with the watchdog

daemon installed in a guest OS, a virtual machine monitor can notice that the daemon is no longer working when periodically trying to communicate with it. Although virtualization technology is widely deployed in PC-based systems, it is difficult to implement it in embedded devices as fewer hardware devices support it. Moreover, since the volume of source code for a virtual machine monitor (VMM) tends to become large, the risk of VMM including bugs also becomes large. To overcome the restriction, Kanda developed SPUMONE, which a lightweight virtual machine monitor designed to work on embedded processors [24]. It provides a function to reboot the guest OS. However, SPUMONE does not provide a memory protection mechanism between the virtual machine monitor and the guest OS (Normal OS). Thus, it is vulnerable to an attack on the virtual machine monitor from the guest OS.

To make a secure environment by utilizing TrustZone, various systems have been proposed.

In [25], Yan-ling et al. propose a secure embedded system environment with multi policy access control mechanism and a secure reinforcement method based on TrustZone. It assumes various applications and services runs on it. In [26], Sangorrin et al. propose a software architecture on which real-time operating system and a general-purpose operating system are executed concurrently on a single ARM processor with low overhead and reliability by utilizing TrustZone. Baseline common functions described in Section IV basically uses the same technique in the existing approaches. Our contribution is clarifying a total architecture and functions which must work in a secure environment with a full implementation to enable end-point devices automatically to recover from an error status in a Smart Grid.

VIII. CONCLUSION AND FUTURE WORK

LiSTEE^{TM} Recovery works effectively to resist critical bugs or attacks including zero-day attacks, that could potentially cause the system to crash, in order to keep availability of end-point devices. The performance evaluation has been presented to show that the degradation of the existing system is sufficiently small. Considering commercialization, we have shown that the development cost and production cost can be minimized. Moreover, LiSTEE^{TM} Recovery can save maintenance cost.

Future work includes resistance to sophisticated attacks. In one possible attack, an attacker illegitimately modifies the shared memory area to fake as if Normal OS works correctly while almost all Normal OS functions actually stop. As a result, LiSTEE^{TM} TA misunderstands that Normal OS works correctly. One approach to solve this attack is to implement LiSTEE^{TM} TA so that it itself checks the status of Normal OS without the support of an application program running on Normal OS. For example, whenever Normal OS is running, it must update a certain data area, such as page tables or process tables. Therefore, in monitoring the data area, LiSTEE^{TM} TA can determine whether Normal OS is working or has crashed. An advantage of LiSTEE^{TM} is that it is impossible for an attacker to reverse-engineer and to tamper with an algorithm of LiSTEE^{TM} TA from Normal OS because of the memory protection mechanism. Thus, an attacker cannot know how to compromise Normal OS in order to produce misleading information. We have not implemented this though. Another possible attack involves damaging the file system locating Normal OS. Network boot can be a solution where LiSTEE^{TM} TA downloads a small rescue program from the head-end system when booting fails.

REFERENCES

[1] Isozaki, H., Kanai, J., Sasaki, S. and Sano. S., “Keeping High Availability of Connected End-point Devices in Smart Grid,” In Proc. Fourth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, Apr. 2014, pp. 73-80.

[2] C4 Security. “The Dark Side of the Smart Grid,” [Online]. Available: http://www.c4-security.com/The Dark Side of the Smart Grid – Smart Meters (in)Security.pdf [Accessed 20 Nov. 2014]

[3] Forsberg, D., Ohba, Y., Patil, B., Tschofenig, H., and Yegin, A., “Protocol for carrying authentication for network access,” IETF RFC 5191 [Online]. Available: http://tools.ietf.org/html/rfc5191 [Accessed 20 Nov. 2014]

[4] Zhao, F., Hanatani, Y., Komano, Y., Smyth, B., Ito, S., and Kambayashi, T., “Secure authenticated key exchange with revocation for smart grid,” The third IEEE PES Conference on Innovative Smart Grid Technologies (ISGT 2012), IEEE Power & Energy Society (PES), Jan. 2012, pp. 1-8.

[5] Wang, W. and Lu, Z., “Cyber security in the Smart Grid: Survey and challenges,” Computer Networks, Vol. 57, Issue 5, Apr. 2013, pp. 1344-1371.

[6] Khurana, H., Hadley, M., Ning, L., and Frincke, D.A., "Smart-grid security issues," IEEE Security & Privacy, Vol. 8, Issue 1, Jan-Feb. 2010, pp. 81-85.

[7] Li. K, "Towards Security Vulnerability Detection by Source Code Model Checking," Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, Apr. 2010, pp. 381-387.

[8] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., and Leu, T., "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)," Computer Security Applications Conference, 2004. 20th Annual, Dec. 2004, pp. 82-90.

[9] Huang, Z and Harris, I.G., “Return-oriented vulnerabilities in ARM executables,” IEEE 2012 Conference on Technology for Homeland Security, Nov. 2012, pp. 1-6.

[10] De Craemer, K., and Deconinck, G., "Analysis of state-of-the-art Smart Metering communication standards,” in Young Researchers Symposium (YRS), 2010. [Online]. Available: https://lirias.kuleuven.be/bitstream/123456789/265822/1/SmartMeteringCommStandards.pdf [Accessed 20 Nov. 2014]

[11] Wang, W., Xu, Y., and Khanna, M., "A survey on the communication architectures in smart grid," Computer Networks, Vol. 55, Issue 15, Oct. 2011, pp. 3604-3629.

[12] Liotta, A., Geelen, D., van Kempen, G., and van Hoogstraten, F., “A survey on networks for smart-metering systems,” International Journal of Pervasive Computing and Communications, Vol. 8, No.1, 2012, pp. 23-52.

[13] Varghese, S. M., and Jacob, K. P., "Anomaly Detection Using System Call Sequence Sets," Journal of Software, Vol. 2, No. 6, Dec. 2007, pp. 14-21.

[14] Pont, M. and R. Ong., “Using watchdog timers to improve the reliability of single-processor embedded systems: Seven new patterns and a case study,” In Proc. First Nordic Conf. on Pattern Languages of Programs, Sept. 2002, pp. 159-200.

[15] ARM. “ARM Security Technology,” [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-

genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf [Accessed 20 Nov. 2014]

[16] Alves, T. and Felton, D., "TrustZone: Integrated Hardware and Software Security," Information Quarterly, Vol. 3, No. 4, 2004, pp. 18-24.

[17] MITRE. "Common vulnerabilities and exposures," [Online]. Available: http://cve.mitre.org [Accessed 20 Nov. 2014]

[18] ARM. "Dhrystone Benchmarking for ARM Cortex Processors," [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dai0273a/DAI0273A_dhrystone_benchmarking.pdf [Accessed 20 Nov. 2014]

[19] Miyashita, M. and Ohtani, T., "Transmission Characteristics Evaluation of Demand-side Communication -Evaluation of Response Time Using International Standard Protocol for Meter Reading and Wireless LAN-," CRIEPI Research Report, Jun. 2011 (in Japanese).

[20] Kleen, A., "Machine check handling on linux," Technical report, SUSE Labs, Aug. 2004 [Online]. Available: http://halobates.de/mce.pdf [Accessed 20 Nov. 2014]

[21] Dolev, S. and Yagel, R., "Towards Self-Stabilizing Operating Systems," IEEE Transaction on Software Engineering, Vol. 34, No. 4, Jul/Aug. 2008, pp. 564-576.

[22] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D., "Terra: A virtual machine-based platform for trusted computing," In Proc. Symposium on Operating System Principles, Oct. 2003, pp. 193-206.

[23] "libvirt - the virtualization API.," [Online]. Available: http://libvirt.org [Accessed 20 Nov. 2014]

[24] Kanda, W., Yumura, Y., Kinebuchi, Y., Makijima, K., and Nakajima, T., "SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems," In Proc. Embedded and Ubiquitous Computing, Dec. 2008, pp. 144-151.

[25] Yan-ling, Z., Wei, P., "Design and Implementation of Secure Embedded Systems Based on Trustzone," In Proc. International Conference on Embedded Software and Systems, Jul. 2008, pp. 136-141.

[26] Sangorrin, D., Honda, S. and Takada, H., "Dual Operating System Architecture for Real-Time Embedded Systems," In Proc. 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Jul. 2010, pp. 6-15.