# A Study on Security and Data Aggregation for M2M Gateway Systems

**2016, September**

**Yuichi Nakamura**

**Graduate School of Natural Science and Technology (Doctor's Course) OKAYAMA UNIVERSITY**

# Contents

**1 Introduction** **1**

1.1 Background . . . . . 1

1.1.1 M2M gateway in IoT system . . . . . 1

1.1.2 Functions of M2M gateways . . . . . 2

1.1.3 Constraints of M2M gateways and research purpose . . . . . 4

1.2 Research problems . . . . . 5

1.2.1 Security technologies . . . . . 5

1.2.2 Data aggregation function . . . . . 7

1.3 Related work . . . . . 9

1.3.1 Techniques supporting SELinux policy creation . . . . . 9

1.3.2 Application of SELinux for embedded devices . . . . . 11

1.3.3 Data aggregation . . . . . 12

1.4 Research Strategies . . . . . 13

1.5 Outline of the Dissertation . . . . . 15

**2 An approach to facilitate SELinux policy creation** **17**

2.1 Introduction . . . . . 17

2.2 Problems in creating security policy . . . . . 17

2.2.1 Overview of SELinux . . . . . 17

2.2.2 Difficulties in writing security policy . . . . . 19

2.2.3 Overview of refpolicy . . . . . 21

2.2.4 Problems in creating security policy using refpolicy . . . . . 21

2.3 Approach to creating security policy . . . . . 24

2.3.1 Higher level language: SPDL . . . . . 24

2.3.2 SPDL tools . . . . . 26

2.4 Design and implementation of SEEdit . . . . . 26
2.4.1 Design of SPDL . . . . . 27
2.4.2 Implementation of SPDL converter . . . . . 30
2.4.3 Implementation of SPDL tools . . . . . 31
2.5 Evaluation . . . . . 32
2.5.1 Experimental setup . . . . . 32
2.5.2 Result and consideration . . . . . 33
2.6 Conclusions . . . . . 37

**3 Reducing Resource usage of SELinux with OSS contributions 39**
3.1 Introduction . . . . . 39
3.2 Problems in applying SELinux to embedded systems . . . . . 39
3.2.1 Resource usage . . . . . 39
3.2.2 Acceptability to the OSS community . . . . . 42
3.3 Overall approach of Embedded SELinux . . . . . 42
3.3.1 Reducing resource usage . . . . . 43
3.3.2 Modifications acceptable to OSS community . . . . . 45
3.4 Implementation of Embedded SELinux . . . . . 45
3.4.1 Kernel Tuning . . . . . 46
3.4.2 Modification of userland programs . . . . . 48
3.4.3 Preparing policy from scratch . . . . . 49
3.5 Evaluation . . . . . 49
3.5.1 Target device and software . . . . . 49
3.5.2 Benchmark results . . . . . 50
3.5.3 Regressions . . . . . 52
3.5.4 Results of OSS community proposals . . . . . 54
3.6 Conclusions . . . . . 55

**4 A Rule-based Sensor Data Aggregation Framework 57**
4.1 Introduction . . . . . 57
4.2 Issues in aggregation of sensor data . . . . . 57
4.2.1 Categories of functions in the process of sensor data aggregation on an M2M gateway . . . . . 58

4.2.2 Issues with sensor data aggregation using firmware programming . . . 60
4.3 Proposal of Complex Sensor Data Aggregator . . . . . . . . . . . . . . . . . . 63
4.3.1 Basic design of the CSDA . . . . . . . . . . . . . . . . . . . . . . . . 63
4.3.2 Design of static aggregation processor . . . . . . . . . . . . . . . . . . 64
4.3.3 Configuration language . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.3.4 Update module . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.4.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.4.2 Evaluation of advantage of CSDA . . . . . . . . . . . . . . . . . . . . 73
4.4.3 Evaluation of overhead . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**5 Conclusions** **80**
5.1 Concluding remarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.2 Future directions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

**Acknowledgements** **84**

**References** **85**

# List of Figures


1.1 Usage of M2M gateways in IoT service systems . . . . . . . . . . . . . . . . . 2
1.2 Functions of M2M gateways and targeted functions . . . . . . . . . . . . . . 3
1.3 Structure of data aggregation function of M2M gateway . . . . . . . . . . . . 8
1.4 Target problems and approaches . . . . . . . . . . . . . . . . . . . . . . . . 14

2.1 Main feature of SELinux: TE (Type Enforcement) . . . . . . . . . . . . . . . 18
2.2 Main components of SELinux policy language . . . . . . . . . . . . . . . . . 19
2.3 Example usage of attribute . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.4 Example of an m4 macro . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.5 Part of the configuration for the ftp daemon in refpolicy . . . . . . . . . . . 22
2.6 A configuration example of SPDL for ftp daemon . . . . . . . . . . . . . . . 25
2.7 The architecture of SEEdit . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.8 Statements in SPDL to allow access to resources . . . . . . . . . . . . . . . . 29
2.9 SPDL configurations and output of SPDL converter . . . . . . . . . . . . . . 30
2.10 An example of permission mapping . . . . . . . . . . . . . . . . . . . . . . . 32
2.11 Template generator GUI to generate typical configurations . . . . . . . . . . 33
2.12 Template generator GUI to generate using knowledge of users . . . . . . . . 34

4.1 Overview of M2M system . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.2 Architecture of the CSDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3 Design of Static Aggregation Processor . . . . . . . . . . . . . . . . . . . . . 65
4.4 Array usage of sampling buffer . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.5 Part of configuration for input step . . . . . . . . . . . . . . . . . . . . . . . 70
4.6 Part of the configuration for data processing step . . . . . . . . . . . . . . . 71
4.7 Part of the configuration for output step . . . . . . . . . . . . . . . . . . . . 72
4.8 Comparison of C and CSDA at input step . . . . . . . . . . . . . . . . . . . 75

# List of Tables

1.1 Fit & gap analysis of protection methods for M2M gateways . . . . . . . . . . 6
1.2 Related work about security policy creation . . . . . . . . . . . . . . . . . . . . 9
1.3 Related work about application of SELinux for embedded devices . . . . . . 11
1.4 Related work about data aggregation . . . . . . . . . . . . . . . . . . . . . . . . 12

2.1 Typical process of creating a security policy . . . . . . . . . . . . . . . . . . . 26
2.2 Systems used in the evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.3 Number of permissions in refpolicy and SPDL . . . . . . . . . . . . . . . . . . 36
2.4 Comparison of number of configuration lines . . . . . . . . . . . . . . . . . . . 36
2.5 Result of resource consumption in the embedded system . . . . . . . . . . . 37

3.1 System call execution time and overhead of SELinux on an embedded system 41
3.2 Read/write execution time and overhead on an embedded system . . . . . . 41
3.3 Files related to SELinux . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.4 Features included in SELinux userland . . . . . . . . . . . . . . . . . . . . . . 44
3.5 SELinux commands ported to BusyBox . . . . . . . . . . . . . . . . . . . . . . 48
3.6 Read/write system call execution time measured by LMbench . . . . . . . . 51
3.7 Read/write execution time on the evaluation board measured by UnixBench 51
3.8 File size increase related to SELinux . . . . . . . . . . . . . . . . . . . . . . . 52
3.9 Memory usage by SELinux . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.10 Open/close execution time and overhead of SELinux on the evaluation board 54
3.11 Effect of reducing number of hash slot . . . . . . . . . . . . . . . . . . . . . . 54
3.12 OSS versions where the proposed modifications are merged . . . . . . . . . . 55

4.1 Embedded system used in the evaluation . . . . . . . . . . . . . . . . . . . . . 74
4.2 Memory usage of CSDA varying configuration of sampling buffer . . . . . . . 77

# Summary

With the growth of machine-to-machine (M2M) and big data technologies, more and more devices are connected to the Internet. As a result, the Internet of Things (IoT) services that provide new value by utilizing sensor data from devices are rapidly emerging. In order to bridge devices in industrial fields and social infrastructures to the Internet, a device called M2M gateway is necessary. In addition to basic bridge functions, security and data aggregation of M2M gateways are necessary for IoT services as an increase of connected devices. IoT is being applied for critical systems such as electric grid and health care, and impact of attacks to such systems are serious. Since M2M gateways are entrance to fields from the Internet and easily exposed to attacks, security for M2M gateways must be considered. The cost of communication between M2M gateways and servers and the cost of server resources will also increase rapidly, because data traffic is expanding year by year. To reduce the traffic, the importance of data aggregation increases, too.

In developing security and data aggregation functions, three constraints specific to M2M gateways must be considered, i.e. hardware resource limitation, difficulty in updating programs and various development environments. However, existing security and data aggregation technologies are not enough considering these constraints. From security perspective, Security-Enhanced Linux (SELinux) is an effective protection method because it works without updates, on various system configurations; resource usage does not increase after deployment. However, there are two problems in applying SELinux to M2M gateways. First problem is difficulty in creating security policy. In configuring SELinux policy, developers must understand lots of permissions and configuration elements such as labels. Additionally, they should also pay attention to the size of policy. In this dissertation, a SELinux configuration tool with a higher level language is proposed to facilitate policy creation process. Second problem is basic resource usage of SELinux. SELinux developers have primarily focused on enterprise server implementation, and resource consumption has become unac-

Division of Industrial Innovation Sciences
Graduate School of Natural Science and Technology

ceptable for M2M gateways. They have to be tuned and modified codes must be merged into codes maintained by Open Source Software (OSS) communities to ensure its long-term use. A tuning method of SELinux for M2M gateways acceptable for OSS communities is proposed. From data aggregation perspective, sensor data are condensed into statistic values and filtered in M2M gateways by dedicated C programs embedded in firmware. However, there are problems in developing and updating C programs. It is difficult to program in C because resource constraints and various development environments should be considered; updating the logic is also difficult because of risky firmware update. For these problems, a rule-based sensor data aggregation framework is proposed.

To address issues of SELinux security policy creation, a security policy configuration system called SELinux Policy Editor (SEEdit) is proposed. SEEdit facilitates creating security policies by utilizing a higher level language called Simplified Policy Description Language (SPDL) and SPDL tools. SPDL reduces the number of permissions by integrated permissions and removes label configurations. SPDL tools generate security policy configurations from access logs and tool user's knowledge about applications. Evaluation results on M2M gateway environment show that SEEdit can create practical security policies.

In order to reduce resource usage of SELinux, Embedded SELinux is proposed. Embedded SELinux is composed of three techniques of reducing resource usage. Firstly, the Linux kernel is tuned to reduce CPU overhead and memory usage. Secondly, unnecessary codes are removed from userland. Thirdly, security policy size is reduced with SEEdit. Embedded SELinux was evaluated using an evaluation board targeted for M2M gateways, and results show that its read/write overhead is almost negligible, and file space requirement and RAM usage are acceptable. In addition, to facilitate acceptance by OSS communities, side effects of the modified codes were carefully measured, and side effects were not observed. As a result, the modified codes were successfully merged into OSS communities.

A sensor data aggregation framework called Complex Sensor Data Aggregator (CSDA) is proposed to support developing and updating aggregation logic on M2M gateways. The functions comprising the data aggregation process are subdivided into the categories of filtering, statistical calculation, and concatenation. The proposed CSDA supports this aggregation process in three steps: the input, periodic data processing, and output steps. The behaviors of these steps are configured by an XML-based rule. The rule is stored in the data area of flash ROM and is updatable through the Internet without the need for a firmware update.

In addition, in order to keep within the memory limit specified by the M2M gateway's manufacturer, the number of threads and the size of the working memory are static after startup, and the size of the working memory can be adjusted by configuring the sampling setting of a buffer for sensor data input. The proposed system was evaluated in an M2M gateway experimental environment. Results show that developing CSDA configurations is much easier than writing dedicated programs in C and the performance evaluation demonstrates the proposed system's ability to operate on M2M gateways.

As a result of the above research, SELinux has become applicable to M2M gateways, and security of M2M gateways can be enhanced. Furthermore, creating and updating aggregation logic is facilitated, and it will contribute to communication and server cost reduction. Finally, the effectiveness of the presented work is also proved by the fact that they are used in commercial systems and also create basis of subsequent researchers' work.

Division of Industrial Innovation Sciences
Graduate School of Natural Science and Technology

# Chapter 1

# Introduction

## 1.1 Background

### 1.1.1 M2M gateway in IoT system

With the growth of machine-to-machine (M2M) infrastructures and big data technologies, more and more devices are connected to the Internet and data are gathered. M2M infrastructures have evolved from traditional private line based networks which are expensive, to the Internet based infrastructures. Consequently, it has become easier to send data from devices to remote servers. For example, a market forecast shows that number of devices connected to the Internet will reach more than 10 billion in 2020 [1]. From data processing perspective, big data technologies such as Hadoop [2] and Spark [3] enable processing huge amounts of gathered sensor data from devices. As a result of such advance of technologies, Internet of Things (IoT) services are rapidly emerging. IoT services provide new value by utilizing sensor data gathered from devices to servers over the Internet. For instance, construction and agricultural machine manufacturers provide remote monitoring and predictive maintenance services to reduce down time for their products by sending sensor and location data from products in the field to servers over the Internet [4, 5, 6]. In production lines of factories, sensor data gathered to servers are also utilized to enhance productivity [7, 8]. Moreover, there are IoT services for consumers such as health care and home automation [9, 10].

In order to enable IoT services, a device called *M2M gateway* is necessary as a bridge between servers on the Internet and devices in fields, as shown in Figure 1.1. Sensors and

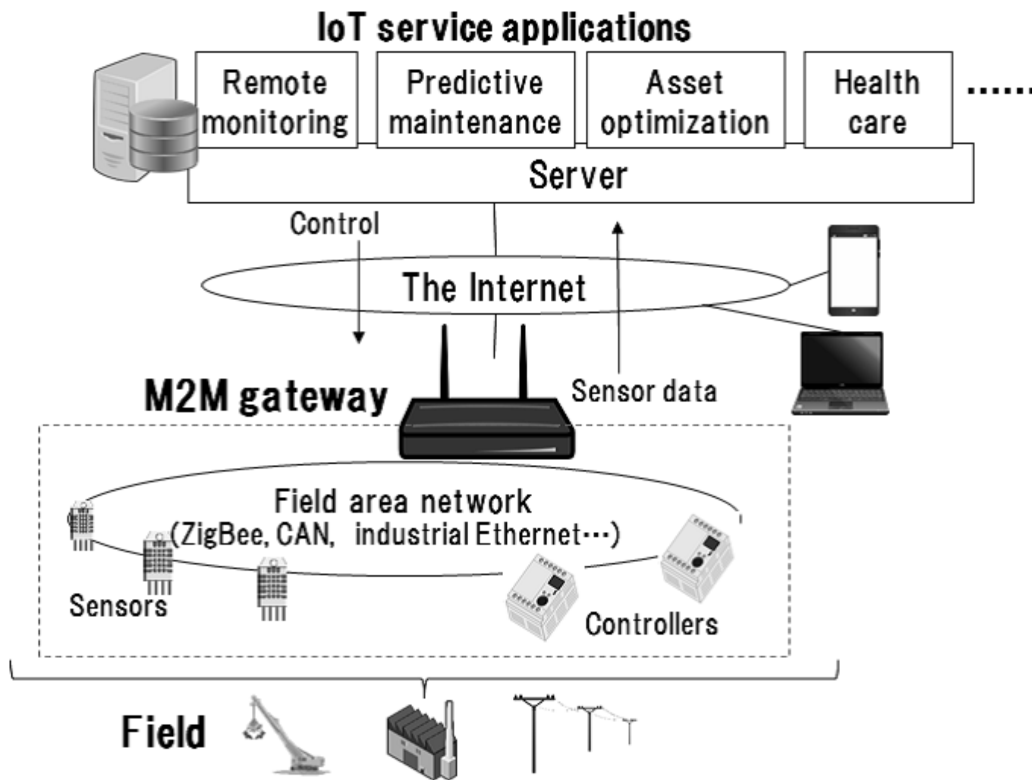


Figure 1.1 Usage of M2M gateways in IoT service systems

controllers in fields are connected to field area networks such as ZigBee [11], controller area network (CAN) [12] and industrial Ethernet [13]. They also cannot reach to servers in the Internet by themselves. Therefore, they have to be bridged to the Internet to gather their data to servers. An M2M gateway is able to communicate with both sensor network protocols and the IP [14]. For example, when using a remote monitoring service for construction machinery, an M2M gateway is attached to the machine and it gathers data from the sensors via the CAN. Then it uploads data to a server via a wireless Internet connection.

### 1.1.2 Functions of M2M gateways

Figure 1.2 shows functions of M2M gateways. As described, the basic function of M2M gateways is a bridge of field and the Internet. The bridge is composed of three functions: *WAN communication*, *data conversion* and *field area network communication*. The WAN communication function is responsible for sending data and receiving control signal from servers. This function is usually based on existing standard such as TCP/IP and 3GPP

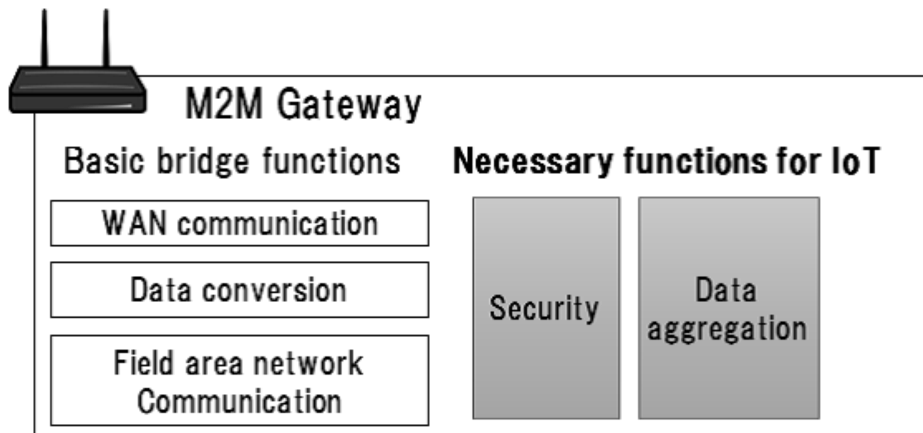


Figure 1.2 Functions of M2M gateways and targeted functions

to ensure interoperability. The data conversion function translates data formats between WAN and field area network protocols. The field area network communication function is responsible for field area network communication between M2M gateways and sensors. For interoperability, it also utilizes standardized protocols such as ZigBee and CAN.

In addition to these bridge functions, security and data aggregation functions are necessary for IoT services as an increase of connected devices. The security function prevents attacks from the Internet, and the data aggregation function condenses raw sensor data such as taking statistical values. From security perspective, IoT is being applied for critical systems such as electric grid and health care, and the impact of attacks to such systems is serious [15]. Since M2M gateways are entrance to fields from the Internet and easily exposed to attacks, security for M2M gateways must be considered. Security of Linux systems is particularly important, because OSs of M2M gateways are shifting from dedicated real-time OSs to Linux where vulnerabilities are widely shared and malwares effective to M2M gateways were found [16]. From data aggregation perspective, cost of communication between M2M gateways and servers and cost of server resources will increase rapidly, because data traffic will increase. For example, it is estimated that data traffic will be 1,000% increase from 2014 to 2020 [17]. To reduce the traffic, the importance of data aggregation also increases. From above background, security and data aggregation functions for M2M gateways are focused as research areas of this dissertation.

### 1.1.3 Constraints of M2M gateways and research purpose

In the development of security and data aggregation functions, following three M2M gateway specific constraints must be considered.

#### Constraint 1: Resource limitation

Since M2M gateways for industry and social infrastructure usage are expected to work for a long time, mature hardware is preferred. In addition, M2M gateways are often battery powered and energy consumption must be small. Consequently, CPU and memory resources for M2M gateways are limited compared with enterprise server systems. CPU clock is around 200-800 MHz, RAM size is around 2-512 MB. In industrial usage, they tend to be especially limited, i.e., CPU is 200-600 MHz, RAM size is around 2-64 MB. For example, RAM size for an M2M gateway to monitor construction machines [18] is only 2-8 MB. Second example is an industrial controller with M2M gateway functionality [19] to monitor production lines. RAM size for user programs is 8-16 MB. Another example can be found in a general purpose industrial communication board, with a CPU clock of 250 MHz and 64 MB of RAM size [20].

#### Constraint 2: Difficulty in program update

Updating programs is difficult because programs of M2M gateways are usually stored in read-only flash ROM file systems. Firmware update where entire file system image is re-written, is necessary in order to update programs. However, updating firmware is risky because devices will not be usable if it fails. Furthermore, the cost of delivering firmware image becomes large when a network contract is measured-rate, or when updated by human, the cost would be raised more. Therefore, it is desirable to minimize events of firmware updates.

#### Constraint 3: Various development environments on Linux

Linux is assumed as an OS of M2M gateways, because Linux is the most popular OS for embedded systems. For example, a market research reports that more than 60 % of embedded systems are based on Linux in 2015, and the rate will increase 15 % every year

[21]. However, other parts of M2M gateway environment are various. For example, CPU architecture varies depending on gateways, e.g. ARM, SH, x86, and x64. The development environment, such as compilers and tool chains, also strongly depends on M2M gateways because M2M gateways are not standardized. It is desirable for security and data aggregation functions to work on various environments.

Considering above, the research purpose was set as providing security and data aggregation technologies suitable for M2M gateways that meet these three constraints.

## 1.2 Research problems

In this section, research problems in the dissertation are described. Firstly, two problems are introduced in the security function, then one problem is introduced in the data aggregation function.

### 1.2.1 Security technologies

Attackers and malwares exploit vulnerabilities of M2M gateways, then they destroy systems, steal information, and attack other systems. Table 1.1 shows fit & gap analysis of protection methods considering three constraints of M2M gateways introduced in section 1.1.3.

For enterprise servers, applying security patches and anti-virus software are effective protection methods. However, they are impractical for M2M gateways considering constraints. For applying security patches, constraint 1 is met because it does not require additional resources as long as firmware update function is included. However, constraint 2 is not met because it assumes firmware update. Constraint 3 is not met because device vendors often customize embedded Linux systems, using them in place of standard Linux distributions. Device-specific security patches are not typically provided by Linux distributions. For anti-virus software, all constraints are not met and is not suitable for M2M gateways. Constraint 1 is not met because pattern files are usually black-list based and the size increases as new attacks are found. Constraint 2 is not met because updating scan engines requires a firmware update. Constraint 3 is not also met, various system configurations must be considered in preparing pattern files.

Table 1.1 Fit & gap analysis of protection methods for M2M gateways

| | Constraint 1<br>Resource usage | Constraint 2<br>Difficulties in update | Constraint 3<br>Various development environment |
|---|---|---|---|
| Application of security patch | Met | Not met | Not met |
| Anti-virus software | Not met | Not met | Not met |
| Exploit execution protection | Met | Met | Not met |
| SELinux | Partially met | Met | Met |

Exploit execution protection technologies [22, 23] are more suitable for M2M gateways, but they are not enough. These techniques protect against attacks that exploit memory management vulnerabilities, for example by randomizing stack and heap layouts, and implementation for embedded systems is available [24]. They meet constraint 1 and 2 because code size is tiny and work without firmware update. However, constraint 3 is not met, because they are specific to CPU architecture, and not available for some CPU architecture. In addition, they only protect against attacks that exploit memory management vulnerabilities, and techniques to bypass them have been reported [25, 26].

Mandatory Access Control (MAC) is useful for mitigating intrusion [27]. With MAC, every access to resources is checked based on a set of rules called security policy, and no user, including root, can avoid the check. If attackers obtain the root privilege, their behavior can still be confined by the security policy, and attack attempts will fail owing to lack of access rights. Various MAC systems are provided for Linux on top of Linux Security Modules (LSM) [28], such as SELinux [29], TOMOYO [30] and SubDomain [31]. The most representative implementation is SELinux, because SELinux provides the most fine-grained access control model, and SELinux is the most widely used and actively maintained implementation. Furthermore, SELinux is suitable for M2M gateways considering constraints. Constraint 2 and 3 are met, because it works even if security patches are not applied [32], and does not depend on CPU and userland. Constraint 1 is partially met, because the se-

curity policy is white-list based and the size does not increase after new attacks are found, but the basic size of SELinux should be small.

As described above, SELinux is effective for M2M gateways. However, to apply SELinux to M2M gateways there are two problems.

#### Problem 1: Creating security policy of SELinux

First problem is creating security policies. Creating security policies is difficult because access rules often exceed 10,000. In addition, elements in rules such as permissions, labels and macros are more than 1,000 and they are understandable only to SELinux experts. In order to facilitate policy writing, a sample policy is prepared by a community for enterprise server usage, but there are no sample policy for M2M gateways. Consequently, a lot of customization is required for M2M gateway usage.

#### Problem 2: Resource usage of SELinux

Second problem is basic resource usage of SELinux. SELinux developers have primarily focused on enterprise server usage, and many features have been added. As a result, resource consumption has become unacceptable for embedded systems such as M2M gateways, and they have to be tuned. In the tuning, modified codes must also be merged into codes maintained by OSS communities to ensure their long-term use. OSS codes are made public by various open-source licenses; developers from around the world continuously submit patches, which are reviewed and merged into the OSS communities' codes according to their development processes [33]. If modified codes are not merged into OSS community codes, the developer must correct the modified codes for every version up of the target OSS. Conversely, if modifications are merged, they are maintained by OSS communities and the modified code will be used for an extended period.

In the dissertation, applying SELinux to M2M gateways is aimed through proposal of technologies for these two problems.

### 1.2.2 Data aggregation function

Figure 1.3 shows the structure of a typical data aggregation function. The data aggregation function is implemented in the aggregation logic. The aggregation logic takes raw sensor data

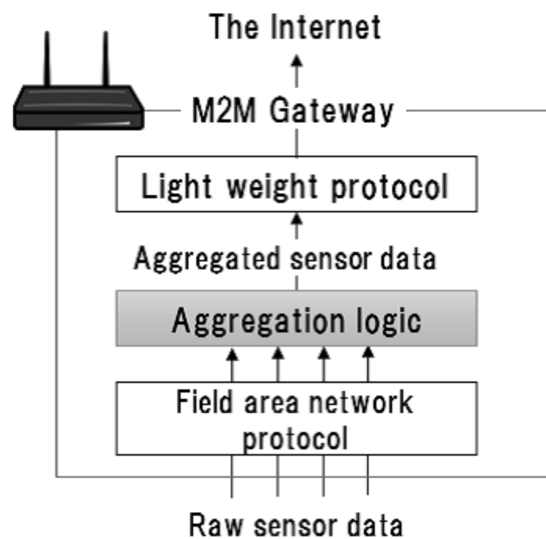


Figure 1.3 Structure of data aggregation function of M2M gateway

from device drivers of M2M gateways, and the quantity of data is reduced by taking such as distributions and averages [34, 35, 36]. Then the aggregated data are sent to the Internet through light weight protocols to reduce protocol overhead [37, 38, 39, 40].

For such data aggregation logics, dedicated programs are usually developed and installed in firmware of M2M gateways. However, dedicated programs are not suitable for IoT service providers because there are difficulties in developing and updating aggregation logics considering three constraints of M2M gateways.

**Problem 3: Difficulties in developing and updating aggregation logics**

Firstly, developing dedicated aggregation logics is difficult for IoT service providers with consideration for constraints of M2M gateways. For resource constrained environment, the programming language for firmware development is usually C. Programming in C requires bothersome memory management, an issue that is hidden in higher-level languages. In addition, IoT service providers need to learn the development environment, such as compilers and tool chains, which strongly depends on the target M2M gateway because M2M gateways are not standardized. Attention also needs to be given to the CPU and RAM usage of the aggregation program, particularly the matter of RAM usage. The input data rate cannot be predicted, and sufficient memory must be allocated in advance. However, when the input rate increases suddenly and too much memory is allocated, the system will crash because M2M

Table 1.2 Related work about security policy creation

| Category | Related work |
|---|---|
| Initial creation | • Sample policy [43, 44] |
| Verification | • Visualization [45, 46]<br>• Formal analysis [47, 48, 49] |
| Deployment | • Modularization [50]<br>• Policy delivery server [50, 51, 52] |
| Modification | • Generation from log [53, 54]<br>• Template generation [55]<br>• Policy removal [56] |
| All the above | • IDE [57, 58, 59]<br>• Security policy description language [60, 61] |

gateways often do not have swap. Consequently, it is difficult for M2M service providers to write aggregation programs on M2M gateways.

The second difficulty arises when changing the data aggregation logic on the firmware. IoT service providers need to change aggregation logics in their Plan Do Check Act (PDCA) cycle [41, 42]. In order to change the data aggregation logic, the firmware must be updated. However, firmware updates are risky as described in constraint 2.

In this dissertation, developing and updating data aggregation logics on M2M gateways are facilitated for IoT service providers.

## 1.3 Related work

In this section, work related to above three problems is reviewed.

### 1.3.1 Techniques supporting SELinux policy creation

There are a lot of methods to support SELinux policy creation. Table 1.2 summarizes related work categorized by policy development process: initial creation, verification, de-

ployment, and modification.

The most famous method to support initial policy creation is sample policy [43, 44] developed by the SELinux community. The purpose of sample policy is to remove needs of describing configuration by preparing access rules for typical applications. This approach works well for enterprise server usage based on famous Linux distributions, because there are many community developers. However, sample policy is not suitable for embedded devices such as M2M gateways because system configurations and applications are different depending on devices, and there are few community developers.

There are several methods related to security policy verification [45, 46, 47, 48, 49]. SETools [45] and SEGrapher [46] provide visualization methods to check whether security policies are properly configured. Tools to help formal analysis of security policies are proposed [47, 48, 49]. They input an information flow model created from a security goal, then proves whether the security policy achieves the security goal or not.

In the deployment phase, basic technology is semodule [50]. Semodule provides modularity to SELinux security policies: version management and install/update/removal of security policy modules. On top of semodule, servers that deliver security policies to multiple machines have been proposed [50, 51]. A server to deliver security policy module for an application with the application install/update was also proposed [52]. These servers are useful, but do not solve the difficulty in writing configuration.

In the modification phase, audit2allow [53] and setroubleshoot [54] help to generate additionally necessary configurations from logs for an existing security policy. Polgen [55] generates template configurations for additional applications. A policy removal method from an existing policy is also proposed [56]. For M2M gateway systems, security policies have to be created before using such tools.

Integrated Development Environment (IDE) and security policy description languages are useful throughout the process of policy development. SLIDE [57] and CDS Framework [58] are IDEs that have text editors specialized for SELinux security policies and launchers to deploy a SELinux security policy to the target system. SELinux/AID [59] has also a text editor. Additionally, it has host based Intrusion Detection System (IDS) based on SELinux log. However, basic difficulties in configuring SELinux policies are not solved in these IDEs. Permissions, labels and macros still have to be handled. Security policy description languages are also proposed [60, 61] to make writing security policies easier than macros in sample

Table 1.3 Related work about application of SELinux for embedded devices

| Category | Related work |
|---|---|
| Application of basic functions | • Porting of basic components [62, 63]<br>• Tuning with hardware assistance [64] |
| Integration with userland programs | • Integration with Android [65, 66]<br>• Building applications utilizing SELinux [67, 68, 69] |

policy. These languages enhance modularity and reuse of templates, but do not solve the problem of permissions and labels.

On the above existing methods, they do not provide a method to create security policies for M2M gateways where resource is limited and system configurations are various.

### 1.3.2 Application of SELinux for embedded devices

Table 1.3 summarizes work related to application of SELinux for embedded devices.

There are several reports about applying basic functions of SELinux to embedded systems. The first work was reported by Coker [62]. In the work, SELinux features in Linux kernel and SELinux userland were tuned and ported to an ARM based hand held device. However, the SELinux implementation used here is old whose version is before merged to mainline Linux. SELinux implementation had been changed a lot when it was merged to mainline Linux; LSM [28] is used for example. Vogel et.al. [63] ported SELinux to another mobile platform. The work is basic porting and performance issue is not addressed. About performance tuning for embedded systems, hardware assistance to reduce overhead of SELinux for embedded systems is proposed [64]. However, cost is a problem in this approach. A special purpose processor is required and implementation of SELinux on the processor will increase software development cost.

On top of basic SELinux functions, there is userland work. Shabtai et.al. [65] and Smalley et.al. [66] used SELinux for Android platform. The main focus of them is enabling SELinux access control in Android userland framework layer. They extend the SELinux access control model to Android specific userland permissions. Systems on top of SELinux for embedded

Table 1.4 Related work about data aggregation

| Category | Related work |
|---|---|
| Development framework | • Application of OSGi framework [70, 71, 72, 73]<br>• Application of DSMS [74, 75, 76]<br>• Distributed aggregation [77, 78, 79] |
| Data process method | • Compression [80, 81, 82]<br>• Filtering [83] |

systems are proposed [67, 68, 69]. Simple information flow models were implemented [67, 68] and trusted boot was implemented [69]. SELinux is used as a tool to achieve their goals.

The above related work does not solve the problem of resource consumption of SELinux for M2M gateways. SELinux and its userland were tuned in several existing reports, but modifications were not merged to communities' codes and are not usable for the current version of SELinux.

### 1.3.3 Data aggregation

Table 1.4 shows related work about data aggregation. They are categorized into overall development frameworks and individual data processing methods.

There are a lot of systems about development frameworks that are applicable to M2M gateways. They are sub-categorized into three items. Firstly, OSGi [70] is used to facilitate writing programs for M2M gateways [71, 72]. Li et.al. [73] also proposed an OSGi-based data collection framework including data aggregation on gateway devices. OSGi is Java based and it is suitable for writing complicated applications such as home energy management system. However, proprietary Java is assumed and the cost is raised. In addition, memory usage often becomes large such as more than 10 MB. Therefore, applicable M2M gateways are limited for OSGi. Secondly, a data stream management system (DSMS) [74] partially removes coding for data aggregation. It processes the input data stream based on the rule called query. However, because it is not originally intended for sensor data aggregation, the rule language does not cover the input and output steps. It is primarily intended for real-time processes, so it usually requires enough CPU and memory resources. On the other

hand, there is a DSMS for resource-constrained devices [75], but its rule are embedded in the program and a firmware update is necessary to change that. Yamamoto and Koizumi proposed a DSMS for distributed environment on top of MySQL [76]. It facilitates describing data aggregation logics by utilizing SQL, but it uses traditional ring buffer for statistical calculation and not suitable for memory constrained environments. Thirdly, distributed data aggregation [77, 78, 79] techniques are proposed. A language interpreter is installed in each sensor nodes including gateways, and only necessary data are sent from sensor nodes, thereby reducing the quantity of data sent from the M2M gateways. They are suitable for controlling devices within sensor networks because their response times are fast: when a sensor detects an event, an action is immediately issued from the sensor. However, they are not enough for usage of reducing the quantity of data transferred between M2M gateways and servers for two reasons. First, these systems are not intended for processing data from multiple sensors. It is efficient to gather a sensor's data depending on another sensor value; however, these technologies do not support such complex processes. Second, the installation and management costs are substantial. Data processing middleware and configurations need to be installed on all sensor nodes.

Data processing methods such as data compression and filtering are proposed. For data compression, Yamaguchi et al. [80] developed a compression algorithm on M2M gateways by utilizing multi-dimensional similarity of sensor data. Papageorgiou et al. [81] developed an algorithm to choose the best data compression method for sensor data aggregation. Matamoros et al. [82] aggregated sensor data on M2M gateways to reduce overhead of bridging sensor to the Internet. About data filtering, Papageorgiou et al. [83] also proposed a threshold filtering technique for specific situations. They focus on data compression and filtering on specific situations.

The above related work does not solve the problem of developing and updating various aggregation logics on resource constrained M2M gateways.

## 1.4 Research Strategies

This section explains the research strategy in the dissertation. Fig 1.4 summarizes target problems mentioned in section 1.2.1 and 1.2.2, and approaches in the research.

Firstly, a security policy configuration system is proposed for the problem of creating

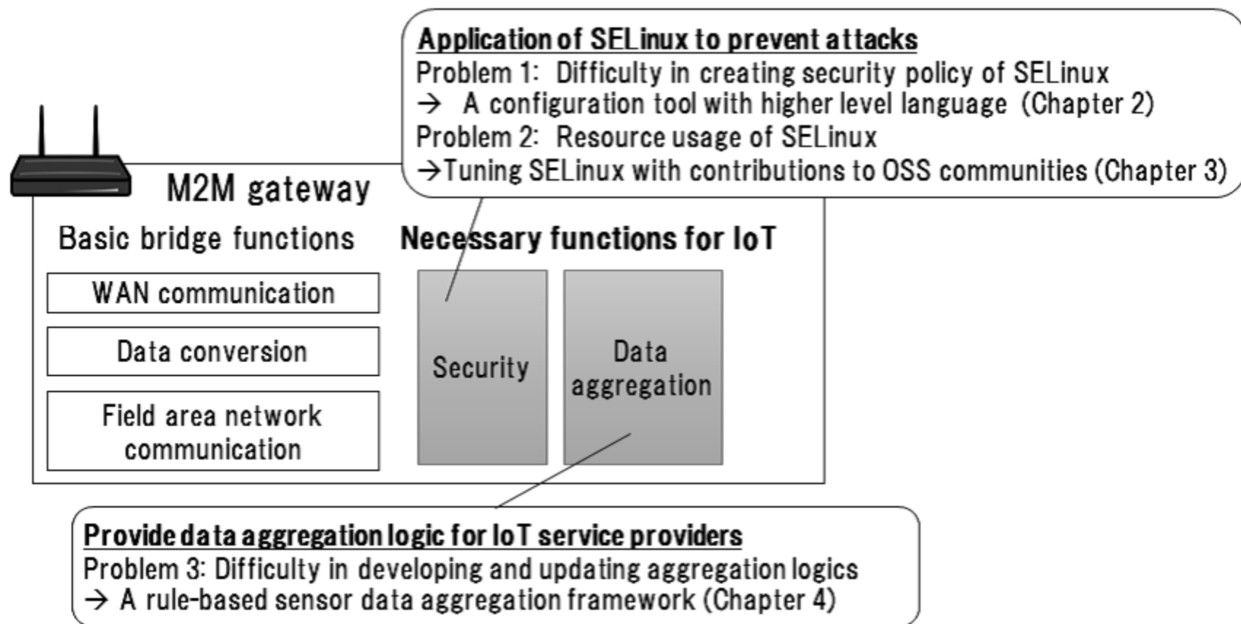


Figure 1.4 Target problems and approaches

SELinux security policies. The most popular method to create security policies is customizing Reference Policy (refpolicy) [44]. However, it is difficult to describe configurations in refpolicy because there are more than 700 permissions, 1,000 macros and more than 100,000 lines of configurations. In addition, type labels must be associated with file names and network resources. Furthermore, resource consumption of refpolicy is large because configurations for various applications are included. To remove such difficulties, a security policy configuration system called SELinux Policy Editor (SEEdit) is proposed. SEEdit facilitates creating security policies by utilizing a higher level language called Simplified Policy Description Language (SPDL) and SPDL tools. SPDL reduces the number of permissions by integrated permissions and removes type label configurations. SPDL tools generate security policy configurations from access logs and tool user's knowledge about applications. The proposed system was evaluated on an experimental environment targeted for M2M gateways. In the evaluation, the policy creation process, lines of configuration, and size of security policy were compared with those of refpolicy based policy creation.

Secondly, Embedded SELinux is proposed for the problem of resource usage of SELinux. SELinux developers have primarily focused on enterprise server implementation, and many features have been added. As a result, the resource consumption has become unacceptable for embedded systems. To reduce resource consumption, Linux kernel and related OSS

codes must be modified. In addition, modified codes must be merged into codes maintained by OSS communities to ensure their long-term use. Proposed Embedded SELinux reduces resource usage by using three techniques. First, the Linux kernel is tuned to reduce CPU overhead and memory usage. Second, unnecessary codes are removed from userland libraries and commands. Third, security policy size is reduced by SEEdit. The resource usage of Embedded SELinux was measured in an embedded system evaluation board. In addition, to be acceptable to the OSS community, the side effects of the modified code were also measured. Eventually, modified codes were proposed to related OSS communities.

Thirdly, a rule-based sensor data aggregation system called the Complex Sensor Data Aggregator (CSDA) is proposed for the problem of developing and updating aggregation logics on M2M gateways. An aggregation logic is typically programmed in the C language and embedded into the firmware. However, developing aggregation programs is difficult for M2M service providers because it requires gateway-specific knowledge and consideration of resource issues, especially RAM usage. In addition, modification of the aggregation logic requires the application of firmware updates, which are risky. The proposed CSDA enables IoT service providers to develop data aggregation logics without programming in C. The CSDA defines the framework that supports the sensor data aggregation. It splits the aggregation process into input, data processing, and output steps, whose behaviors are controlled by a configuration file. The aggregation logic can also be changed by simply updating the configuration file and restarting the CSDA. In addition, to work in memory-constrained environments, a static aggregation processor where working memory size and number of threads are fixed is launched at startup time, and the working memory size can be adjusted by configuring a sampling setting. In order to see the effectiveness of the proposed system, the system is compared with developing dedicated programs in C on an M2M gateway experimental environment.

## 1.5 Outline of the Dissertation

The reminder of this dissertation is organized as follows.

Chapter 2 proposes a SELinux policy creation tool called SEEdit. After problems in creating security policies with a popular method are discussed, the proposed approach to facilitate policy creation is introduced. Next, the detailed design and evaluation results are

shown.

Chapter 3 proposes Embedded SELinux to reduce SELinux resource usage. Firstly, problems are introduced by showing resource usage measurement results of existing SELinux. After that, the proposed approach to reduce resource usage and implementation of Embedded SELinux are described. Finally, results of evaluation and proposal to OSS communities are shown.

Chapter 4 proposes a framework called CSDA to facilitate developing and updating data aggregation logics on M2M gateways. At first, the data aggregation process on an M2M gateway is categorized and issues in developing and updating the aggregation logic are discussed. In the next, basic design and detailed design of CSDA is described. Finally, CSDA is compared with dedicated C programs by an experiment on an evaluation board.

Chapter 5 concludes this research and shows directions for the future.

# Chapter 2

# An approach to facilitate SELinux policy creation

## 2.1 Introduction

A security policy configuration system called SEEdit SELinux Policy Editor (SEEdit) is proposed to resolve the problem of creating security policy of SELinux. After detail of problems in existing policy creation method, the approach and design of SEEdit are described. Finally, SEEdit is evaluated on an experimental environment targeted for servers and M2M gateways.

## 2.2 Problems in creating security policy

In this section, problems in creating a security policy for a target system based on refpolicy are described after an overview of SELinux and difficulty in SELinux policy.

### 2.2.1 Overview of SELinux

The primary feature of SELinux is a MAC referred to as type enforcement (TE) [84]. An overview is shown in Fig 2.1. Processes are identified by labels called *domain*, and resources are identified by labels called *type*. By default, a domain cannot access types. A domain is only able to access types that are explicitly specified in the security policy. In the

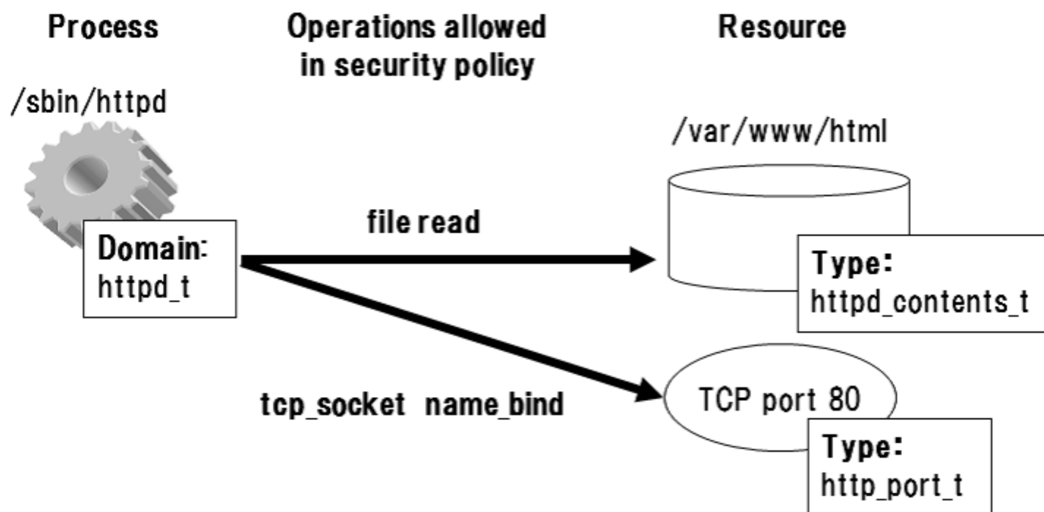


Figure 2.1 Main feature of SELinux: TE (Type Enforcement)

example shown in Fig 2.1, the label *httpd_t* t is assigned to the http daemon process, the label *http_content_t* is assigned to /var/www, and the label *http_port_t* is assigned to TCP port 80. In the security policy, http_t is granted permission related to file reading and port binding. As a result, the http daemon is allowed to read /var/www and bind TCP port 80; no other access is permitted, unless described in the policy. These access control functions are implemented in the Linux kernel using LSM hook functions to perform security checks during system calls [85]. In addition, userland programs are necessary to manage the security policy and labels. For example, when the security policy is changed, the kernel must reload it.

The security policy is loaded to SELinux kernel in binary representation. However, it is hard to handle the binary security policy because it is unreadable for humans. To represent the security policy in text, SELinux has a basic policy language [86] that is mainly composed of four syntax elements shown in Figure 2.2.

SELinux identifies resources by labels called *type*. Types are assigned to resources such as files, port number and NICs (Figure 2.2(1)). Domains and types must be declared by *type* statements(Figure 2.2(2)). <type or domain> inherits configurations described for <attribute>. Statements in 2.3 are examples of attributes. admin_t can read both httpcontent_t and ftpcontent_t.

The *allow* statements (Figure 2.2(3)) comprise access rules, i.e. domains are permitted to

```
(1) Type assignment
<file name>  system_u:object_r:<type>
portcon <port number> system_u:object_r:<type>
netifcon <NIC name> system_u:object_r:<type>
(2)  Label declaration
type <type or domain>, <attribute>;
(3) Allowing access
allow <domain> <type> <permission>;
(4) Conditional policy expression
if(<parameter>){<statement>}
```

Figure 2.2 Main components of SELinux policy language

```
type httpcontent_t, content;
type ftpcontent_t, content;
allow admin_t content:file read;
```

Figure 2.3 Example usage of attribute

access types using several permissions. <permission> is composed of *object classes* and *access vector permissions*. Object class means classification of resources such as *file* (normal file), *dir* (directory) and *tcp_socket* (TCP socket). For each object class, access vector permissions such as *read* and *write* are defined. For example, permission *file read* means reading normal files, *dir read* means reading directories. Conditional policy expression(Figure 2.2(4)) is written to support multiple use cases in one security policy. <statement> is effective only when <parameter> is true. For instance, when CGI is necessary, the parameter *httpd_enable_cgi* is set true, then rules related to using CGI are enabled.

### 2.2.2 Difficulties in writing security policy

Detailed access control in SELinux is effective in preventing tactics used by attackers but poses problems when writing security policy.

### Amount of access rules

To grant enough permissions for applications to work correctly, a lot of access rules must be configured because the SELinux security policy is white list that means applications are not granted any permissions unless rules are described. In addition, more than 700 permissions make the amount much larger. In fact the total number of access rules in a security policy is often more than 10,000.

### Describing individual access rules

Access rules are composed of *allow* statements, but it is hard for engineers to write individual rules because of permissions and types.

(1) Configuring permissions
Since SELinux permissions are designed from the view point of system calls, engineers are required to have knowledge of the Linux kernel. For example, when engineers want to grant an application to use TCP ports. Permissions related to system call such as *listen*, *accept*, *bind* and so on should be configured. Additionally, the number of permissions which exceed 700 makes using permissions more difficult.

(2) Configuring types
Engineers have to assign types to all files and ports in a system. This means that for each file they have to group certain files and ports as needed and then name types and assign them. The work is very hard because there are more than 10,000 files in standard Linux systems. There are also two more difficulties with types. Firstly, engineers have to get used to types because they have been identifying files by file names not types in traditional Linux. Secondly, there is also a problem of dependency in assigning new types. This problem is explained with an example. When the foo_t type is assigned under /foo directory and the bar_t domain is allowed to read the foo_t type, the bar_t domain can read all files under the /foo directory. If the foo2_t type is newly created, and assigned to the file /foo/foo2, then the bar_t domain cannot access /foo/foo2 because the bar_t domain is not allowed to access foo2_t. In this way, the bar_t domain was able to read /foo/foo2 before assigning the new type foo2_t, but bar_t cannot access /foo/foo2 after the new type is assigned to /foo/foo2.

```
#Macro definition
define(‘r_file_perms’,‘file { read getattr lock ioctl }’)
#Example usage of the macro
allow httpd_t contents_t r_file_perms;
```

Figure 2.4 Example of an m4 macro

### 2.2.3 Overview of refpolicy

Creating a security policy from scratch is therefore unrealistic in view of the above difficulties. The most popular way to facilitate creating security policy is *refpolicy* [44] which is developed and maintained by the SELinux community. Refpolicy is composed of macros and configurations for typical applications.

(1) Macros
M4 [87] macros are defined to describe frequently used phrases in short words. For instance, Figure 2.4 shows *r_file_perms* macro, which is expanded to permissions related to reading regular files.

(2) Configurations for typical applications
Configurations for applications shipped with Linux distributions are prepared by the SELinux community and Linux distributors and they are included in refpolicy. Figure 2.5 is a part of the configuration for the ftp daemon. There are many macros, such as *init_daemon_domain*, *miscfiles_read_public_files* and so on. In the figure, conditional expressions are omitted, but in fact, many conditional expressions are also included because refpolicy is intended to support as many use cases as possible such as ftp login, and DB connection.

### 2.2.4 Problems in creating security policy using refpolicy

Customizing refpolicy is necessary when the system usage case or its installed applications is beyond the expectations of refpolicy. For example, embedded systems and commercial applications are not within the scope of refpolicy. However, there are three problems in customizing refpolicy. One is the difficulty in describing configurations, second is the difficulty of verifying refpolicy and third is resource consumption.

```
#Assign ftpd_t domain to ftp daemon
 1 type ftpd_t;
 2 type ftpd_exec_t;
 3 init_daemon_domain(ftpd_t,ftpd_exec_t)
 4 init_system_domain(ftpdctl_t,ftpdctl_exec_t)
 5 /usr/sbin/vsftpd --gen_context(system_u:object_r:ftpd_exec_t,s0)
#Permit ftpd_t to read contents
 6 miscfiles_read_public_files(ftpd_t)
 7 /var/ftp(/.*)? gen_context(system_u:object_r:public_content_t,s0)
#Permit ftpd_t to wait connection on tcp port 21
 8 corenet_non_ipsec_sendrecv(ftpd_t)
 9 corenet_tcp_sendrecv_all_if(ftpd_t)
10 corenet_udp_sendrecv_all_if(ftpd_t)
11 corenet_tcp_sendrecv_all_nodes(ftpd_t)
12 corenet_udp_sendrecv_all_nodes(ftpd_t)
13 corenet_tcp_sendrecv_all_ports(ftpd_t)
14 corenet_udp_sendrecv_all_ports(ftpd_t)
15 corenet_tcp_bind_all_nodes(ftpd_t)
16 corenet_tcp_bind_ftp_port(ftpd_t)
17 corenet_tcp_bind_ftp_data_port(ftpd_t)
18 corenet_dontaudit_tcp_bind_all_ports(ftpd_t)
19 portcon tcp 21  gen_context(system_u:object_r:ftp_port_t,s0)
```

Figure 2.5 Part of the configuration for the ftp daemon in refpolicy

### Difficulty in describing configurations

The major difficulty in describing configurations is complicated configuration elements such as permissions, macros and types. The main reason for the complexity is the number of configuration elements. For example, there are more than 700 permissions and more than 1,000 macros and 1,000 types. In addition, difficulties in configuring types as discussed in section 2.2.2 still remain, and nested macro definitions make understanding macros harder.

### Difficulty in verifying refpolicy

The refpolicy that creates the security policy must be verified to ensure the quality of that security policy. In this context, *verify* means understand what is configured, then find misconfigurations and modify them. However, it is difficult to verify because of the

complexity of the configuration elements as stated before. In addition, the following points make verification more difficult.

- Amount of configurations
  The size of refpolicy makes verification more difficult. For example, refpolicy included in Fedora 9 has configurations for almost all applications shipped with Fedora 9 and is composed of more than 2,000 types and more than 150,000 access rules.

- Conditional expressions
  Many conditional expressions are embedded in refpolicy, and they are sometimes included in macro definitions. Thus, it is difficult to figure out which configurations are enabled.

- Attributes
  Attributes are often used for types and they increase the time necessary to understand what configurations mean, as shown in the next example. The line *allow httpd_t httpd-content:file read;* is included in refpolicy. *httpd_t* is a domain for the apache daemon, and *httpdcontent* is an attribute. To understand what kind of files httpd_t can access from the line, types that have the httpdcontent attribute have to be found by searching for type declaration statements, which are sometimes embedded in macro definitions.

### Resource consumption

A security policy is saved as files in storage, then it is loaded to RAM at system boot. Therefore, the security policy consumes storage and RAM. Since refpolicy is intended for multiple use cases, many conditional expressions and configurations for many applications are included. As a result, the size of refpolicy becomes large. For example the refpolicy included in Fedora Core 6 consumes 1.4 MB storage and 5.4 MB RAM. In resource constrained systems such as embedded systems, this is a problem because they often have less than 64 MB RAM and storage.

### Tools for refpolicy

Tools are developed by the SELinux community to aid customizing refpolicy based security policies.

To help describing configurations, tools called setroubleshoot[54], SLIDE [57] and system-config-selinux [88] are developed. Setroubleshoot analyzes access logs and presents configurations when an application does not work due to SELinux access denial. SLIDE is an Integrated Development Environment (IDE) to configure refpolicy. It has features to aid describing configurations such as input completion. system-config-selinux is a tool to generate templates of configurations for new applications. It can generate templates using a wizard. These tools are helpful, but remain problems. Setroubleshoot is only targeted to handle minor modifications. SLIDE and system-config-selinux do not solve complexities of SELinux policy language and refpolicy such as many permissions, types and macros.

To aid verifying refpolicy, SETools [45] and SEGrapher [46] have features to query and visualize security policy, such as querying what kind of types a domain can access. They are also useful, but problems discussed in section 2.2.4 still exist.

## 2.3 Approach to creating security policy

To encounter problems in creating security policy based on refpolicy, a different approach is proposed. This means that a new security policy configuration system called SEEdit is proposed, instead of using refpolicy. SEEdit aims to facilitate describing configurations, verifying a created security policy and creating a small security policy. The idea of the proposed system is explained in this section.

### 2.3.1 Higher level language: SPDL

The difficulty in describing configurations in refpolicy is caused by the large number of permissions, complicated macros and type configurations. Sophisticated macros can partly solve such problems, such as by creating a small number of macros and removing nested macro definitions. However, type configurations are still necessary in such macros. Instead of macros, a higher level language *Simplified Policy Description Language (SPDL)* is proposed on top of SELinux policy language. SPDL aims to reduce the number of configuration elements by *integrated permissions* where related SELinux permissions are grouped. In addition, SPDL removes type configurations by identifying resources with their names. An example of configuration by SPDL is shown in Figure 2.6. The configured access rules are

```
1 {
# Assign ftpd_t domain to ftp daemon
2 domain ftpd_t;
3 program /usr/sbin/vsftpd;
# Permit ftpd_t to read /var/ftp
4 allow /var/ftp/** r;
# Permit ftpd_t to wait connection on
tcp port 21
5 allowcom -protocol tcp -port 21 server;
6 }
```

Figure 2.6 A configuration example of SPDL for ftp daemon

almost the same as Figure 2.5 but SPDL is simpler. Permissions related to reading files and directories are merged to integrated permission *r* and permissions to wait for connection on ports are merged to *server*. Additionally, names such as /var/ftp and TCP port 21 are used to identify resources and assigning types to resources is not necessary. To apply SPDL configurations, the *SPDL converter* translates these configurations to SELinux policy language, i.e. SPDL converter generates the necessary type configurations, and expands integrated permissions to related SELinux permissions.

The difficulty in verifying refpolicy is caused by two factors. First is the complicated configuration elements such as macros, permissions, attributes and conditional expressions. This complexity is solved by SPDL. Second is that many lines of configurations for access rules for applications not installed in the system and for rules disabled by conditional expressions are included. The proposed approach to solve the problem of many configuration lines is to describe only necessary configurations by SPDL without refpolicy, i.e. write configurations only for applications installed in the target system. Since neither conditional configurations nor configurations for unused applications are included, the number of configuration lines will likely be reduced. This also helps reduce resource usage by the security policy.

Table 2.1 Typical process of creating a security policy

| Step | Necessary work | Available SPDL tools |
|---|---|---|
| 1 | Assign domain | Template generator |
| 2 | Run application | - |
| 3 | Describe allow rules | Allow generator |
| 4 | Check configuration | - |

### 2.3.2 SPDL tools

In order to support writing configurations by SPDL without refpolicy, SPDL tools composed of *template generator* and *allow generator* are proposed. SPDL tools aim to reduce the number of configurations written by hand during the process of creating a security policy.

Table 2.1 shows a typical process of creating a security policy and this process is iterated for each target applications. Configurations to assign a domain to a target application are described as lines 2 and 3 in Figure 2.6 (Step 1). In order to figure out what kind of access rules should be described, access logs are obtained by running the target application (Step 2). Access rules are described using the access logs (Step 3). For example, when an access log entry shows *foo_t domain read accessed filename bar* then an access rule that allows *foo_t* to read *bar* is described. Run the application again and see whether it works correctly (Step 4). If the application does not work correctly, go back to step 2.

Allow generator supports writing configurations allowing access in step 3 of Table 2.1. An approach of audit2allow [53] is adopted to automate describing configurations, i.e. generate configurations that permit access appearing in access logs.

Template generator outputs configurations in step 1 of Table 2.1 by using configurations typical to application categories. For example, most daemon programs require access rights to create temporary files under /var/run and communicate with syslog. To produce more configurations, template generator uses the knowledge of the tool user about the target application, such as what kind of files and network resources the application accesses.

## 2.4 Design and implementation of SEEdit

SEEdit was designed and implemented following the approaches discussed in the previous section. SEEdit provides SPDL and SPDL tools layers on top of SELinux policy language as

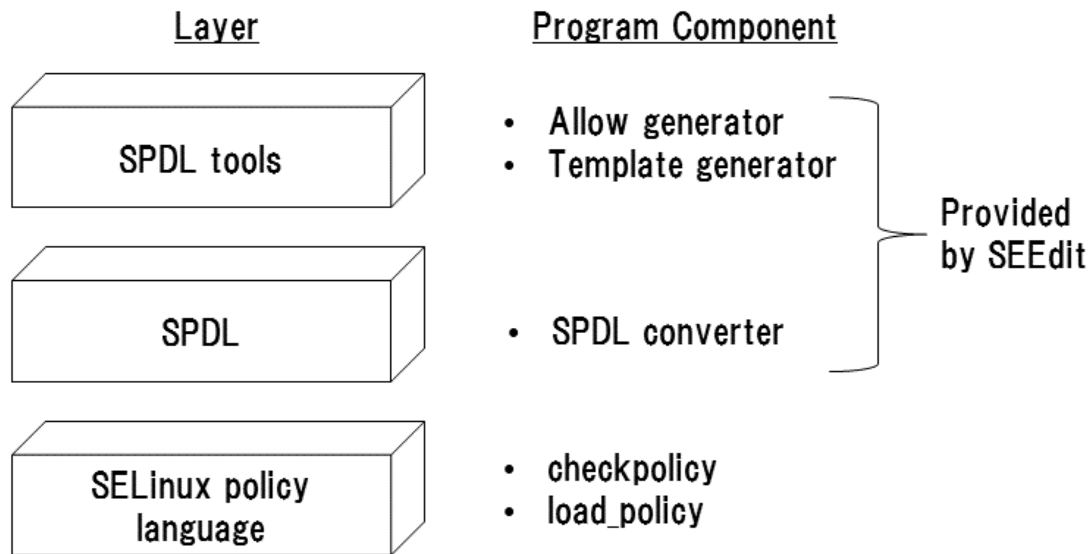


Figure 2.7 The architecture of SEEdit

shown in Figure 2.7. In the SPDL layer, SPDL converter generates the security policy written in SELinux policy language from configuration written in SPDL. SELinux policy language is converted to binary format by *checkpolicy*, then loaded to SELinux kernel by *load_policy* command. In SPDL tools layer, SEEdit has *allow generator* and *template generator* to help writing configuration. The design of SPDL and the implementation of SPDL converter and SPDL tools are described in the following subsections.

### 2.4.1 Design of SPDL

The main features of SPDL are *integrated permissions* to reduce the number of permissions, and configurations using resource names to remove type configurations. SPDL also has an *include* statement to reduce the number of lines. The detail is explained in this section.

#### Integrated permissions

While integrated permissions reduce the number of permissions by grouping permissions, those permissions important for security should be kept. In order to include such important permissions, integrated permissions are designed from the viewpoint of protecting the confidentiality, integrity and availability of a target system. Compromising confidentiality

happens when unexpected information goes out, and compromising integrity happens when unexpected information comes into the system. Thus, permissions related to input and output to files, network resources and IPCs have to be included in integrated permissions. The other permissions are privileges which can be abused to compromise availability and to facilitate attacks. For example, *setrlimit* permission that controls the resource usage limit of processes can lead to compromised availability. Permission *cap_insmod* can result in installation of malicious kernel modules. Therefore, privileges have to be included in integrated permissions. The details of integrated permissions are shown as follows.

(1) Integrated permissions for files
Integrated permissions for files are taken from previous research by Yamaguchi et.al [89] because they are designed to control input and output to files and directories. These integrated permissions are, *r* (read), *x* (execute), *s* (list directory), *o* (overwrite), *t* (change attribute), *a* (append), *c* (create), *e* (erase) and *w* (= o+t+a+c+e).

(2) Integrated permissions for network
Two integrated permissions related to input and output are designed for port numbers, NIC, IP address and RAW socket. For example, integrated permissions for port numbers are *server* (wait for a connection from outside) and *client* (begin a connection to outside).

(3) Integrated permissions for IPC
Integrated permissions for Sysv IPCs are *send* and *recv* to control input and output to processes. Integrated permissions for signals are designed to control sending each signal because SELinux can only control the sending of signals. For example, integrated permission *k* allows sending sigkill.

(4) Integrated permissions for other privileges
46 integrated permissions for other privileges are designed. Almost all permissions about privileges are included to prevent attackers from compromising availability and facilitating attacks. However, overlapped permissions are merged as an exception. For example, SELinux permission *capability net_admin* and *netlink_route_socket nlmsg_write* overlap each other because they are related to change kernel configuration of network. They are therefore merged to the integrated permission *net_admin*.

```
(1) Statements
(a) Permits access to files
    allow <filename> <integrated permission>;
(b) Permits access to network resources
    allownet <resourcename> <integrated permission>;
(c) Permits access to domain using IPC
    allowcom <IPCname> <domain>  <integrated permission>;
(d) Permits usage of privilege
    allowpriv <integrated permission>
(2) Example
(a) Permits to read files under /foo/bar directory.
    allow /foo/bar/** r
(b) Permits to wait connection on tcp port 80.
    allownet -protocol tcp -port 80 server;
(c) Permits to read data from process running as foo_t domain
 via unix domain socket.
    allowcom -unix foo_t r;
(d) Permits to use chroot system call.
    allowpriv cap_sys_chroot;
```

Figure 2.8 Statements in SPDL to allow access to resources

### Configurations using resource names

To remove type configurations, SPDL enables configurations using resource names. SPDL statements *allow* and *allownet* are designed as shown in Figure 2.8 to enable name based configurations for files and network resources such as port number, NIC and IP address. Configurations for resources where type configuration is not required are also supported. In IPCs and other privileges, Assigning types for IPCs and privileges is not required in SELinux. For such resources, *allowcom* and *allowpriv* are also designed as shown in Figure 2.8.

```
(1) SPDL to be converted by SPDL converter
1 domain ftpd_t;
2 allow /var/ftp/** r;
(2) Output of SPDL converter
# Declare and assign type
1 type var_ftp_t;
2 /var/ftp(|/.*) system_u:object_r:var_ftp_t
#Allows permissions related to integrated permission r
3 allow ftpd_t var_ftp_t:lnk_file { iotcl lock read  };
4 allow ftpd_t var_ftp_t:file { iotcl lock read };
5 allow ftpd_t var_ftp_t:fifo_file { iotcl lock read };
6 allow ftpd_t var_ftp_t:sock_file { iotcl lock read };
```

Figure 2.9 SPDL configurations and output of SPDL converter

**Include statement**

In order to reduce the number of configuration lines, the *include* statement imports configuration from a file. For example, when the file daemon.te includes access rules commonly used for daemon applications, describing *#include daemon.te;* imports those access rules.

### 2.4.2 Implementation of SPDL converter

SPDL converter translates SPDL to SELinux policy language. The translation process is shown with an example of converting SPDL configurations in Figure 2.9(1) to configurations in Figure 2.9(2).

The ftpd_t domain is allowed to read files and directories under /var/ftp in Figure 2.9(1). SPDL converter generates types from resource names. For example, it generates var_ftp_t type from filename /var/ftp, then outputs configuration to assign var_ftp_t under /var/ftp in the first two lines in Figure 2.9(2), and generates configuration to allow access to the generated type as line 3-6 in Figure 2.9(2).

When different types are generated for files or directories under /var/ftp then accesses to such types is allowed. For example, when a domain is configured *allow /var/ftp/content1/***

*r;*, then configuration that assigns var_ftp_content1_t to /var/ftp/content1 is generated. SPDL converter also generates configuration for ftpd_t that allows reading var_ftp_content1_t.

However, configurations using resource names do not work well for files dynamically created by processes. Here, the term, *dynamically created files* means files that are removed and created again. In SELinux, when a file is removed and created again, the type of the file is the same as the directory where it belongs. This behavior is sometimes a problem. For example, *allow /tmp/foo r;* is configured in foo_t domain. At first, /tmp/foo is assigned tmp_foo_t type, but when /tmp/foo is removed and created again, then the type is tmp_t. Therefore, the foo_t domain can no longer access /tmp/foo. To handle such cases, SPDL has *allowtmp* to configure assigning types correctly. The syntax is as follows.

```
allowtmp -dir <directory> -name <type> <integrated permission>;
```

This means files created under <directory>are assigned <type>. When <type> is *auto*, type is named automatically. For example, when foo_t domain creates temporary files under /tmp, *allowtmp -dir /tmp -name auto r;* in foo_t domain have to be described, then type *foo_tmp_t* is generated and assigned to temporary files.

### 2.4.3 Implementation of SPDL tools

#### Allow generator

Allow generator outputs configurations that permit access recorded in the access log. The process is explained by an example below. First, allow generator reads SELinux access log, then extracts domain, resource name and permission from an access log entry. When a log entry is recorded that says *httpd_t domain process accessed filename /foo/bar whose type is foo_bar_t with permission file read*, httpd_t, /foo/bar/ and *file read* is extracted. The extracted information is inadequate for creating SPDL based configuration, because the permission is not an integrated permission. In order to obtain an integrated permission, *allow generator* converts SELinux permissions to integrated permissions by permission mapping, which contains mapping of integrated permission to SELinux permissions as illustrated in Figure 2.10. In the example, recorded SELinux permission is *file read*, then permission mapping is looked up and corresponding integrated permission *file_r* meaning integrated permission *r* for file is found. As a result, allow generator is able to output SPDL based

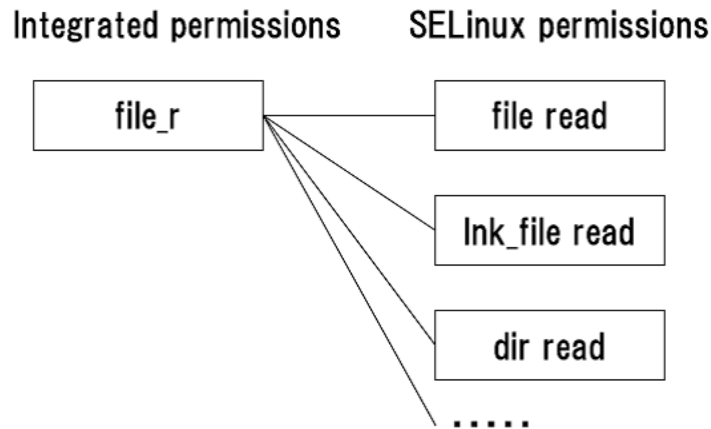


Figure 2.10 An example of permission mapping

configurations *allow /foo/bar/ r;*, from obtained domain, resource name and integrated permission.

#### Template generator

Template generator is implemented as a GUI. Figure 2.11 is a GUI to generate typical configurations. Users choose the profile of applications, and configurations are generated based on the profile. Figure 2.12 is a GUI to generate configurations from user knowledge. They can input their knowledge to the template generator without having to manually enter the SPDL.

## 2.5 Evaluation

### 2.5.1 Experimental setup

In order to make sure the effectiveness of SEEdit, two typical systems were used for experiment. One is an embedded system and the other is a PC system. The embedded system is a low power consumption small server often used for M2M gateways and home gateways, that serve as http and ftp server. It also has portmap daemon for the purpose of sharing files from PC via NFS. The PC system is intended to represent a PC server that

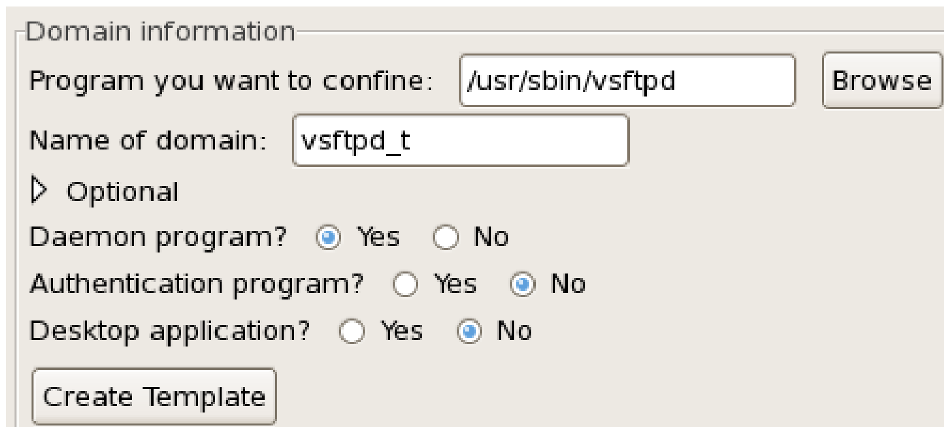


Figure 2.11 Template generator GUI to generate typical configurations

serves typical Internet server applications (HTTP, FTP, Name server, E-mail, File sharing). Table 2.2 shows these systems in detail.

Five domains are configured for services running on the embedded system, 16 domains are configured for services on the PC system. Access rules are written for these services to work properly. Memory usage of the security policy on the embedded system was also measured to evaluate whether SELinux is applicable to embedded systems. The memory consumption by SELinux was defined as the difference between memory usage when SELinux enabled and that when SELinux is disabled.

For comparison, refpolicy based security policies were also evaluated. Three kinds of policies were prepared. The first is default refpolicy shipped with CentOS 5 that was used without modification. Second and third are refpolicy tuned for the PC system and the embedded system. The tuning is assumed to be possible for a usual engineer who is not SELinux specialist. In the tuning, configurations for unused applications are removed by hand, but further removal is not done because only SELinux experts can understand and remove unnecessary configuration lines from each application's configurations.

### 2.5.2 Result and consideration

In the experiment, security policies have successfully been created for both the embedded and the PC system. The process of describing configurations, verifying configurations and resource consumption are reviewed and considered. Lastly, trade-offs in SEEdit are also

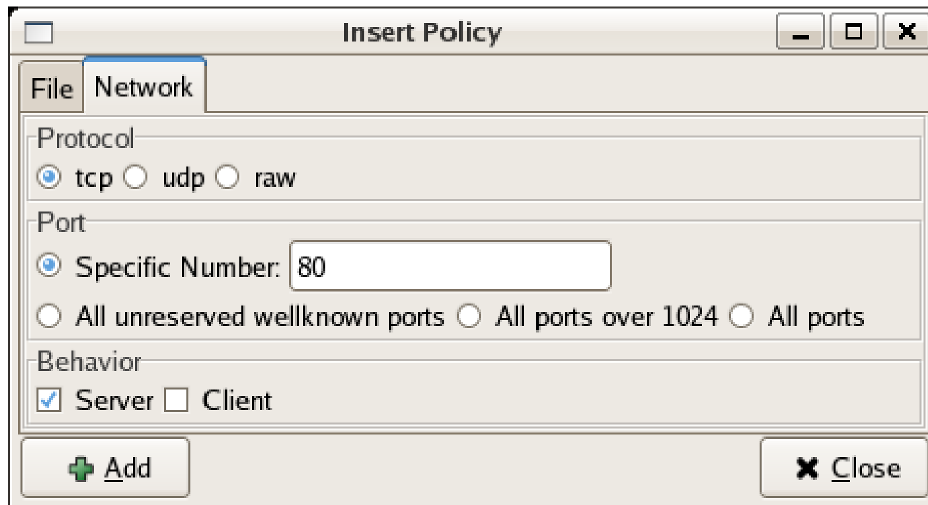


Figure 2.12 Template generator GUI to generate using knowledge of users

discussed.

#### Describing configurations

The first step in describing the configuration is using *template generator*. To evaluate template generator, the assumption of knowledge on the part of the tool user is necessary because generated configurations depend on the user's knowledge. For evaluation, it is assumed that users know how to manage applications, i.e. they know file path of configuration files for applications, names of log files, names of content files which applications deliver and port numbers for applications. Assuming this, template generator produced 52% of the lines of configuration for the evaluation systems. For example, total 24 lines of configurations were described for http service in the PC system, and 12 lines were generated by template generator.

The next step is to produce configurations from access logs by *allow generator*. Most of the configurations generated by allow generator could be used without modification except for the following two cases. First is allow statements generated for dynamically created files. These allow statements have to be replaced with allowtmp statements. For example, foo_t domain dynamically creates and removes /tmp/foo, then log entry *foo_t domain write /tmp/foo* is recorded. Allow generator outputs *allow /tmp/foo w;* from the log entry. However, it should be replaced with *allowtmp -dir /tmp -name auto w;* as shown in section 2.4.2.

Table 2.2 Systems used in the evaluation

| | Embedded system | PC system |
|---|---|---|
| CPU | SH7751R(SH-4) 240 MHz | Core 2 Duo 2 GHz |
| RAM | 64 MB | 1 GB |
| Storage | 64 MB Flash ROM | 10 GB HDD |
| Linux distribution | Not used | CentOS 5 |
| SELinux | Linux 2.6.22 | Linux 2.6.18 |
| Services | httpd, vsftpd, syslogd, klogd, portmap | auditd, avahi_daemon, crond, cupsd, dhclient, gdm, httpd, klogd, mcstransd, named, ntpd, portmap, samba, sendmail |

Second is configurations generated from log entries that record access to normal files. Allow generator outputs *allow /var/www/index.html r;* for httpd_t from log entry *httpd_t read /var/www/index.html*. When the user knows http_t domain accesses /var/www directory, it is better to permit access to directory like *allow /var/www/** r;*. For the above two cases, the generated integrated permissions still can be used without modification.

As shown above, SPDL tools generate most parts of the configurations. In addition, to modify a generated SPDL configuration is easier than modifying refpolicy because the number of permissions is reduced as shown in Table 2.3, complicated macros are not necessary, and type configurations are removed.

### Verifying configurations

To verify created security policy, the difficulty depends on the number of configuration lines. The number of configuration lines are shown in Table 2.4. Refpolicy has more than 90,000 lines even when it is tuned for evaluation systems. In addition, complicated permissions, macros and types are included. On the other hand, in the experiment, the total lines of configuration are 174 for the embedded system, 401 for the PC system, and they are described with SPDL. Therefore, it is easier to verify configurations in SPDL than configu-

Table 2.3 Number of permissions in refpolicy and SPDL

| | refpolicy | SPDL |
|---|---|---|
| File | 130 | 9 |
| Network | 453 | 14 |
| IPC | 45 | 7 |
| Privilege | 80 | 46 |
| Total | 708 | 76 |

Table 2.4 Comparison of number of configuration lines

| | refpolicy (default) | refpolicy (PC) | refpolicy (embedded) | SEEdit (PC) | SEEdit (embedded) |
|---|---|---|---|---|---|
| Number of lines | 147,218 | 98,915 | 92,019 | 401 | 174 |

rations in refpolicy.

Note that verifying configurations written in SPDL is meaningful as long as the output of SPDL converter is correct. Another work is necessary to ensure the result of SPDL converter. One possible way is a test tool. The tool inputs configurations in SPDL and is run for each domain defined in the configurations, then the tool tries all access patterns to see if only accesses configured in the policy are permitted.

#### Resource consumption

The file size of the security policy in the embedded system is 71 KB and RAM usage is 465 KB. In the system used in the experiment, storage is 64 MB, RAM is 64 MB. The consumption of storage and RAM is less than 1%. Thus, the created security policy is usable for the resource constrained embedded devices.

In addition, the resource consumption is also smaller compared with refpolicy based security policy as shown in Table 2.5. The file size of SEEdit based security policy is about 24% and the RAM usage is 40% of tuned refpolicy.

Table 2.5 Result of resource consumption in the embedded system

| | refpolicy (default) | refpolicy (tuned) | SEEdit |
|---|---|---|---|
| File size (KB) | 1,843 | 302 | 71 |
| RAM usage (KB) | 5,820 | 1,168 | 465 |

**Trade-offs**

There are two usage vs. security trade-offs in SEEdit. The first trade-off is integrated permissions used in SPDL because integrated permissions reduce granularity. For example, integrated permission for file *r* means read permissions for file, symlink and socket file. Therefore, allowing read access to symlink but not to file and directory cannot be configured by *r* permission. To solve this problem, the security policy generated by SPDL converter has to be edited. Another solution is to create a new statement in SPDL that enables configuring SELinux permissions directly. The second trade-off is the audit2allow approach in allow generator. If there is a bug or malicious code in a program, and the program accesses files unnecessary for the program to work correctly, allow generator outputs configurations to permit access to such files. For example, if code that accesses confidential data is embedded in a CGI program by a malicious programmer, then a configuration that permits access to the confidential data is outputted by allow generator after running the CGI. To prevent such a dangerous configuration from being included in the security policy, generated configurations should be checked by the SEEdit user. A tool that evaluates generated configurations would prove useful to assist in checking this process.

## 2.6 Conclusions

Security policy for SELinux is usually created by customizing a sample policy called refpolicy. However, creating security policy based on refpolicy has problems in describing and verifying configurations and in resource consumption.

A security policy configuration system SEEdit was proposed. It makes creating security policy easier with a higher level language called SPDL and SPDL tools. SPDL reduces the number of permissions by integrated permissions, and removes type configurations by name

based configurations. SPDL tools help in writing configuration by generating configurations based on access logs and the knowledge of tool users about applications. Experimental results on an embedded system and a PC system have shown that SEEdit resolves problems in creating security policies and that SEEdit can be used to create a practical security policy.

# Chapter 3

# Reducing Resource usage of SELinux with OSS contributions

## 3.1 Introduction

This chapter proposes Embedded SELinux to resolve the problem of resource usage of SELinux for M2M gateways. At first, problems are described with measurement result of existing SELinux on an embedded system targeted for M2M gateways. Next, the design and implementation of Embedded SELinux are shown. Finally, performance measurement result of Embedded SELinux and result of proposal to OSS communities are described.

## 3.2 Problems in applying SELinux to embedded systems

This section describes the resource usage problems of SELinux for embedded systems, and the requirements for tuning code to facilitate its acceptance by OSS communities.

### 3.2.1 Resource usage

To use SELinux, extra features must be enabled in the Linux kernel and userland programs must be added to the standard Linux system. These increase system call overhead, file size, and memory consumption. To reduce power and costs, the hardware resources of embedded

systems are significantly more constrained than PC server systems. For example, in the case of a Linux based M2M gateway, the CPU clock speed is approximately 200 MHz to 600 MHz, the architecture utilizes ARM and SH, RAM size is around 64 MB, and flash memory is used for storage. As a result, the resource usage of SELinux is not acceptable for such embedded systems.

#### Overhead in system calls

When SELinux is enabled, there is overhead for system calls to check the security policy. SELinux is implemented based on Flux Advanced Security Kernel (FLASK) architecture [90], where such overhead is reduced by a cache mechanism called Access Vector Cache (AVC). In a PC environment, P. Loscocco et al. [29] measured this overhead and concluded that it was insignificant. However, problems are encountered when using SELinux on an embedded system. An example of SELinux overhead measured on an embedded system is shown in Tables 3.1 and 3.2. These measurements were performed using LMbench [91] and UnixBench [92]. Overhead is rate of increase in execution time from SELinux disabled kernel. The embedded system platform was composed of a SH7751R (SH-4 architecture, 240 MHz) processor, Linux 2.6.22 and SELinux security policy whose size is 60 KB and number of rule is just 2,000. In particular, read/write overhead is a problem, because they are executed frequently and the overhead is significant. Overhead of greater than 100% was observed during null reads/writes. Moreover, 16% of the overhead remained when reading a 4096 buffer. A size of 4096 bytes is often used for I/O buffers, because it represents the page size for many CPUs in embedded system architectures such as SH and ARM. Note that in 4096 bytes write performance is much worse than others because filesystem cache is fully occupied. Cache is working in others.

#### File size increase

The kernel and userland file sizes increase when SELinux is ported, because of the components listed in Table 3.3. The increase is approximately 2 MB if SELinux for PCs (the SELinux included in Fedora Core 6) is ported without tuning. However, this increase is not acceptable for embedded systems, because flash ROM with a capacity of less than 32 MB is often used to store the file system. If SELinux consumes 2 MB then this is considered

Table 3.1 System call execution time and overhead of SELinux on an embedded system

| LMbench | SELinux disabled | SELinux | |
|---|---|---|---|
| | ($\mu$s) | ($\mu$s) | overhead |
| Null read | 2.39 | 5.49 | 130.0% |
| Null write | 2.07 | 5.10 | 146.6% |
| Stat | 21.48 | 42.29 | 96.9% |
| Create | 108.18 | 284.98 | 163.4% |
| Unlink | 67.74 | 126.3 | 86.4% |
| Open/close | 32.55 | 62.82 | 93.0% |
| Pipe | 33.56 | 55.96 | 66.8% |
| UNIX socket | 76.12 | 99.80 | 31.1% |
| TCP | 259.52 | 316.58 | 22.0% |
| UDP | 162.76 | 207.85 | 27.7% |

Table 3.2 Read/write execution time and overhead on an embedded system

| UnixBench read/write | SELinux disabled | SELinux | |
|---|---|---|---|
| | ($\mu$s) | ($\mu$s) | overhead |
| 256 byte read | 7.95 | 13.25 | 66.6% |
| 256 byte write | 12.84 | 21.42 | 66.8% |
| 1024 byte read | 12.79 | 17.97 | 40.5% |
| 1024 byte write | 19.25 | 27.69 | 43.9% |
| 4096 byte read | 33.96 | 39.45 | 16.2% |
| 4096 byte write | 806.45 | 781.25 | -3.1% |

excessive.

### Memory consumption

SELinux has data structures in the kernel to load the security policy. On a PC, the memory consumed by the security policy is approximately 5 MB. However, this is also unacceptable

Table 3.3 Files related to SELinux

| Component | Additional features |
|---|---|
| Kernel | The SELinux access control feature, audit, and xattr support in filesystem. |
| Library | libselinux, libsepol, and libsemanage |
| Command | Commands to manage SELinux such as load_policy. Additional options for existing commands, such as -Z option for ls to view file label. |
| Policy file | The security policy |

for embedded systems, because the RAM capacity is often less than 64 MB and swapping is not effective. If SELinux is used for embedded systems, the possibility that memory cannot be allocated increases. If memory cannot be allocated, applications will not work correctly.

### 3.2.2 Acceptability to the OSS community

Code must be modified to reduce the resource usage described above. Modified code must be merged to the related OSS community source tree to obtain benefits from the OSS ecosystem. If not merged, modifications must be ported for every version up through the target OSS. To be merged, modifications must be accepted by the target OSS community. There are two primary requirements for acceptance. First, the modifications must not affect existing functions. Second, the modifications cannot cause side effects related to performance.

## 3.3 Overall approach of Embedded SELinux

To apply SELinux to embedded systems, Embedded SELinux is proposed. The approach for using Embedded SELinux to resolve the problems described in the previous section is described in the following section.

### 3.3.1 Reducing resource usage

The resource usage of SELinux shown in Table 3.3 is reduced as follows.

#### Kernel

The overhead related to system calls, RAM and file size usage are all candidates for tuning. To mitigate system call overhead, overhead in reads/writes is particularly reduced, because the overhead is significant as shown in section 3.2.1. To reduce RAM and file size usage, the simplest method to reduce resource usage is to remove unused features. However, it is difficult to remove features from the kernel, because this will significantly affect existing features. For example, data structures and APIs must be modified, which will also have an effect on userland. Therefore, another approach is adopted, in which overhead and bottlenecks are firstly analyzed, and subsequently the tuning of the kernel is performed.

#### Userland

SELinux userland was intended for PC server usage, therefore many features are unnecessary for embedded systems. File size can be reduced by selecting features that meet the following criteria.

- Access control feature of SELinux works.
- Security policy can be replaced.
  Most problems related to SELinux are caused by lack of policy configurations. To correct such problems, features that enable the user to replace security policy is necessary.

Features in SELinux and userland can be classified as shown in Table 3.4. Features 1, 2, and 3 were selected according to the criteria above. Feature 1 is necessary to use access control, while 2 and 3 are required to replace policy.

In addition to removing unnecessary features, commands are integrated into BusyBox [93]. BusyBox is a tool that is widely used in embedded systems, and allows multiple commands to be executed from a single executable binary file. It reduces the overhead of executable file headers by integrating multiple commands into a single binary executable file. It is expected that merging SELinux-related commands to BusyBox will further reduce file size.

Table 3.4 Features included in SELinux userland

| # | Feature | Related packages |
|---|---|---|
| 1 | Load policy<br>Load security policy file to the kernel | libselinux |
| 2 | Change labels<br>View and change domains and types | libselinux<br>policy coreutils<br>coreutils<br>procps |
| 3 | Switch mode<br>Switch permissive/enforcing mode | libselinux |
| 4 | User space AVC<br>Use the access control feature of SELinux<br>from userland applications | libselinux |
| 5 | Analyze policy<br>Access data structure of policy | libsepol |
| 6 | Manage conditional policy<br>Change parameters of conditional policy feature | libselinux<br>libsepol<br>libsemanage |
| 7 | Manage policy module framework<br>Install and remove policy modules | libsemanage |

**Security policy**

In PC systems, SELinux security policy is usually formulated by customizing a security policy template called refpolicy [44]. Customization is typically not required for PC server systems, because the necessary configurations have already been prepared. In contrast, substantial customization is required for embedded systems. For example, to reduce security policy size, unused configurations must be removed. In addition, because the file tree structure is different from PC systems, security policy must be modified to accommodate the structure. Such customization is difficult, because there are more than 2,000 macros and 1,000 type configurations. Therefore, the security policy is prepared from scratch, without

using refpolicy.

### 3.3.2 Modifications acceptable to OSS community

As discussed in section 3.3.1, the source code of the Linux kernel, SELinux library, and BusyBox must be modified. To be acceptable to the related OSS community, these modifications must not adversely affect existing code. To avoid affecting existing code, the kernel, SELinux library, and BusyBox were modified as follows.

- Kernel
  The modifications do not alter the function of the kernel itself, because only tuning is required. However, there may be side effects that affect performance. For example, if memory usage is tuned, performance may decline. Therefore, modifications were proposed to the community that include an evaluation of possible regressions.

- SELinux library
  The proposed modifications remove features that are unnecessary for embedded implementations; however, these features are necessary for PC usage. To resolve this conflict, build flags in the Makefile and #ifndef preprocessor in the C language are used. The build flag **EMBEDDED** is defined in the Makefile, and unnecessary code is enclosed with #ifndef and #endif. When a build is executed with the **EMBEDDED** flag set to **y**, parameters are defined for #ifndef blocks and unnecessary source code is not compiled. Conversely, when a build is executed and the **EMBEDDED** flag is not set to **y**, this code is compiled.

- BusyBox
  BusyBox has a framework that switches included features on or off according to a build flag. The **CONFIG_SELINUX** build flag is already set to link the selinux library; this flag is used to include SELinux-related commands only when the **CONFIG_SELINUX** flag is enabled.

## 3.4 Implementation of Embedded SELinux

Embedded SELinux was implemented following the approaches discussed in the previous section.

### 3.4.1 Kernel Tuning

The Kernel was tuned to reduce read/write system call overhead and memory usage.

#### Reducing read/write overhead

Bottlenecks were analyzed by considering the read/write flow. A process behaves as follows when it reads from, or writes to, a file.

(1) Open file
A process opens a file and obtains a file descriptor with the **open** system call, with a specified access mode (e.g. read only, write). Linux file permissions and SELinux permissions are checked.

(2) Read/write
Using the file descriptor obtained from the **open** system call, a **read/write** system call is executed to complete the actual read and write operations. The access mode is checked first. If the access mode does not match the operation, the read/write process returns a permission error. Next, SELinux permissions are checked. If the validation succeeds, data is read from, or written to, the opened file. Reads and writes are typically called from applications multiple times once a file is opened.

(3) Close
The file is closed by using the file descriptor, and related resources are released.

By examining the read/write flow, it is evident that the SELinux permission verification in the read/write step is duplicated, because an operation that is different from the one allowed in the open step is denied by the access mode check in the read/write step. For example, assume a process attempts to open a file in read-only mode; read access is allowed and write is not allowed in the SELinux policy. In this case, when the process attempts a write system call, access is denied in the access mode check because the file descriptor obtained in the open step only includes read-only access. Therefore, the SELinux permission check is not necessary in the read/write step. However, there are two exceptional cases where SELinux permission checks are necessary for a read/write step. First, if the security policy is changed between the time the file is opened and the time of the read/write call, permission must be

rechecked for the read/write call to accommodate the change. Second, when the domain or type labels are changed in a similar manner, permission must be rechecked.

To perform SELinux permission checks only in the exceptional read and write cases described above, the following changes were introduced in the kernel.

(1) SELinux
A data structure was added to store the state of the security policy, as well as domain and type labels.

(2) Open system call
A new LSM hook function, **security_dentry_open**, was introduced. In the hook function, the states of the security policy, domain, and type are stored in the data structure introduced in (1). SELinux file permissions are rechecked after the states are stored, to account for any state changes.

(3) Read/write system call
Compares the current status of the security policy, domain, and type with those described in (2). SELinux permission is checked only when a status change is detected.

### Reducing memory usage

To reduce memory usage, unnecessary SELinux data structures were removed from the kernel. The largest data structure in SELinux is the hash table in **struct avtab**. Two avtabs are statically allocated in the Linux kernel, and security policy access rules are stored there. During this process, 32,768 hash slots are prepared for each hash table, which consumes approximately 250 KB. In PC systems, the number of access rules often exceeds 100,000. Conversely, in embedded systems, the number of rules is often lower than 10,000, because embedded systems have significantly fewer installed applications compared to PC systems. Therefore, embedded systems require significantly fewer hash slots. To reduce the number of hash slots, they are allocated dynamically based on the number of access control rules in the security policy; the number of allocated hash slots is one-quarter the number of rules.

Table 3.5 SELinux commands ported to BusyBox

| # | Feature | Commands |
|---|---|---|
| 1 | Load policy | load_policy |
| 2 | Change labels | chcon, setfiles, restorecon<br>-Z option for ls and ps |
| 3 | Switch mode | getenforce, setenforce, selinuxenabled |

### 3.4.2 Modification of userland programs

#### Reducing library size

As described in section 3.3.1, unnecessary features are disabled by the build flag, and only libselinux is used. The detailed modifications to libselinux are as follows.

- Remove libsepol function call
  Usage of libsepol, which is approximately 300 KB in size, is forced because libselinux calls the libsepol function to load the security policy. To remove the dependency on libsepol, the security policy is loaded directly by calling kernel functions, thus bypassing libsepol.

- User space AVC and conditional policy are disabled
  When the **EMBEDDED** build flag is enabled, files related to these functions are not compiled, and code fragments related to them are disabled by the #ifndef preprocessor.

#### Integrating commands to BusyBox

As discussed in section 3.3.1, commands related to Table 3.4 1-3 are ported to BusyBox from the libselinux and policycoreutils packages using the **CONFIG_SELINUX** build flag. Ported commands are shown in Table 3.5. The -Z option logic to show labels when using ls and ps is embedded in ls and ps BusyBox applets. Other commands are also implemented as BusyBox applets. They are compiled only when the **CONFIG_SELINUX** flag is enabled.

### 3.4.3 Preparing policy from scratch

Writing SELinux security policy is difficult because there are hundreds of permissions and label configurations. To facilitate policy writing, SEEdit is adopted, which was described in the previous chapter. SEEdit simplifies security policy configuration using a higher-level language called Simplified Policy Description Language (SPDL). SPDL reduces number of permissions by integrated permission, and hides label configuration by path name based configuration. Security policy is written using SPDL and converted into a kernel-loadable format by SEEdit.

## 3.5 Evaluation

Performance of Embedded SELinux is measured on an evaluation board that is used for development of M2M gateway and CE devices and performance is compared with SELinux without tuning. Possible regression is also measured and the result of proposal to community is shown.

### 3.5.1 Target device and software

The devices and software versions used in the evaluation are listed below.

(1) Target device

A Renesas Technology R0P751RLC001RL (R2DPLUS) board is used as a target device. This board is often used to evaluate software for embedded devices such as M2M gateway and CE devices which are targets of embedded Linux. The specifications are shown below.

- CPU: SH7751R (SH-4) 240 MHz
- RAM: 64 MB
- Compact flash: 512 MB
- Flash ROM: 64 MB (32 MB available for root file system)

SELinux can be ported to both compact flash and flash ROM. For convenience, benchmarks are measured using the compact flash system.

(2) Software and policy

The software versions and policy used in the evaluation are listed below.

- Kernel: Linux 2.6.22
- SELinux userland: libselinux 2.0.27
- Security policy (before tuning): policy.21 and file_contexts file were taken from selinux-policy-targeted-2.4.6-80.fc6, obtained from Fedora 6.
- Security policy(after tuning): Written by SELinux Policy Editor, including configurations for 10 applications. Not all applications are confined similar to targeted policy [94].

### 3.5.2 Benchmark results

Standard SELinux and Embedded SELinux were ported to the target board, and measured benchmarks for both. The benchmark results for read and write overhead, file size, and memory usage are shown below.

#### System call overhead

Read and write system call overhead was measured by LMbench and UnixBench. The same security policy (files size is 60 KB, 2,000 access rules are included) is used to eliminate the effects from differences in policy. The results are listed in Table 3.6 and Table 3.7. The same security policy is used on both cases. Before tuning, the SELinux overhead for read/write was significant. Null read/write overhead was reduced by 90% after tuning. The overhead in reading the 4096 buffer was a significant problem; however, it was mostly eliminated in Embedded SELinux. In addition, Yamamoto et al. [95] described the effectiveness of the proposed modifications when testing them using a Pentium 4 PC.

#### File size

The file sizes related to SELinux are summarized in Table 3.8. Userlands are built with -Os flag and stripped here. As a result of tuning, file sizes were reduced to 211 KB from 2,287 KB. On the evaluation board, the flash ROM available for the root file system is 32

Table 3.6 Read/write system call execution time measured by LMbench

| LMbench | SELinux disabled | Standard SELinux | | Embedded SELinux | |
|---|---|---|---|---|---|
| | ($\mu$s) | ($\mu$s) | overhead | ($\mu$s) | overhead |
| Null read | 2.39 | 5.49 | 130.0% | 2.68 | 12.5% |
| Null write | 2.07 | 5.10 | 146.6% | 2.38 | 14.9% |

Table 3.7 Read/write execution time on the evaluation board measured by UnixBench

| UnixBench | SELinux disabled | Standard SELinux | | Embedded SELinux | |
|---|---|---|---|---|---|
| | ($\mu$s) | ($\mu$s) | overhead | ($\mu$s) | overhead |
| 256 byte read | 7.95 | 13.25 | 66.6% | 9.24 | 16.2% |
| 256 byte write | 12.84 | 21.42 | 66.8% | 16.29 | 26.8% |
| 1024 byte read | 12.79 | 17.97 | 40.5% | 14.46 | 13.1% |
| 1024 byte write | 19.25 | 27.69 | 43.9% | 22.89 | 19.0% |
| 4096 byte read | 33.96 | 39.45 | 16.2% | 35.08 | 3.3% |
| 4096 byte write | 806.45 | 781.25 | -3.1% | 806.45 | 0.0% |

MB. The size of SELinux is less than 1% of this capacity; therefore, it is acceptable for the evaluation board.

Integrating commands to BusyBox reduces overhead by 28 KB. Commands require 4 KB, and seven commands were ported. The effect will increase as the number of ported commands increases.

#### Memory usage

Memory usage was measured with the free command. The usage by SELinux was measured as follows.

A = The result of the free command when the SELinux-enabled kernel was booted;
B = The result of the free command when the kernel without SELinux was booted;
Therefore, memory usage by SELinux = A - B.

Table 3.8 File size increase related to SELinux

| Component | Standard SELinux (KB) | Embedded SELinux (KB) |
|---|---|---|
| Kernel(zImage) | 74 | 74 |
| Library | 482 | 66 |
| Command | 375 | 11<br>Without using BusyBox: 39 |
| Policy file | 1,356 | 60 |
| Total | 2,287 | 211 |

Table 3.9 Memory usage by SELinux

| Component | Standard SELinux (KB) | Embedded SELinux (KB) |
|---|---|---|
| Hash tables in struct avtab | 252 | 1 |
| SELinux program and policy | 5,113 | 464 |
| Total | 5,365 | 465 |

SELinux's memory usage was measured for both standard SELinux and Embedded SELinux. The memory usage of the hash table in **struct avtab** was also measured to see the effects of tuning. Code that displayed the size of the allocated tables was inserted into the kernel for that purpose. The results are shown in Table 3.9. The memory consumption after tuning was 465 KB and the evaluation board's memory capacity is 64 MB. Therefore, SELinux consumes less than 1% of the board's memory capacity. This demonstrates that, the device has ample resources to accommodate SELinux.

### 3.5.3 Regressions

#### Possibility of regressions

The modifications to the SELinux library and BusyBox do not affect existing features, because they are implemented as options. If developers want to use existing features as

before, they only need to set the SELinux build flag to off. However, modifications to the kernel may affect the performance of existing features as described below.

- Tuning of read/write overhead
  This may affect performance when files are opened, because the new LSM hook function **security_dentry_open** was added.
- Tuning of struct avtab
  This may affect performance when the security policy is retrieved, because the hash table size was reduced.

### Measurement of side effect

These possible side effects of the proposed modifications were measured on the evaluation board as follows.

- Tuning of read/write overhead
  The performance of the file opening process was measured by LMbench for standard SELinux and for Embedded SELinux.
- Tuning of struct avtab
  To measure performance of security policy retrievals, the processing time of **security_compute_av**, where security policy is searched from avtab, must be measured. The time required to call security compute av 10,000 times from the startup of SELinux was measured, because the time required to perform a single function call was too short to measure. The time was measured by varying the number of hash slots when the security policy (which contains 8,188 rules) was loaded.

### Result

The results of side effect measurements are shown in Tables 3.10 and 3.11.

- Side effects on the performance of open
  The results of the performance measurements for the open system call are shown in Table 3.10. The security policy used here is the same as Table 3.6. The performance does not decrease in Embedded SELinux; it is actually better than standard SELinux.

Table 3.10 Open/close execution time and overhead of SELinux on the evaluation board

| LMbench | SELinux disabled | Standard SELinux | | Embedded SELinux | |
|---|---|---|---|---|---|
| | ($\mu$s) | ($\mu$s) | overhead | ($\mu$s) | overhead |
| Simple open/close | 32.55 | 62.82 | 93.0% | 58.70 | 80.3% |

Table 3.11 Effect of reducing number of hash slot

| Number of hash slot | longest chain length | time to call security_compute_av (s) |
|---|---|---|
| 8,192 (=number of rules) | 13 | 9.67 |
| 4,096 (=number of rules/2) | 21 | 9.65 |
| 2,048 (=number of rules/4) | 35 | 9.68 |
| 1,024 (=number of rules/8) | 64 | 9.78 |

One possible reason is that instructions generated by the C compiler were arranged more efficiently for Embedded SELinux.

- Side effects on the performance of security policy searches
  The performance of the security policy search is shown in Table 3.11, when number of SELinux access rules is 8,188. No drop in performance was observed, even when the number of allocated hash slots was reduced to one quarter the number of access rules.

To summarize, no adverse side effects were caused by the proposed modifications.

### 3.5.4 Results of OSS community proposals

The proposed modifications were submitted to the Linux, BusyBox, and SELinux OSS communities, using the diff utility [96] to highlight the modifications; evaluation results were

Table 3.12 OSS versions where the proposed modifications are merged

| Modification | Merged OSS version |
|---|---|
| Reducing read/write overhead | Linux 2.6.24 |
| Reducing size of avtab | Linux 2.6.24 |
| Reducing size of libselinux | libselinux 2.0.35 |
| Integrating SELinux commands | BusyBox 1.9.0 |

included as well. The modifications were successfully merged, as shown in Table 3.12. The links to the patches can be accessed on the Embedded Linux Wiki at `http://elinux.org/SELinux`.

## 3.6 Conclusions

Applying SELinux to embedded devices such as M2M gateways and CE devices where CPU is around 200 MHz to 600MHz and RAM size is often around 64 MB, presents several challenges. The SELinux kernel, userland, and the security policy all consume an unacceptable amount of hardware resources. Related code must be tuned, and tuned codes must be merged into OSS community code to benefit from usage on the its ecosystem.

Embedded SELinux was developed in which resource usage is reduced, using code modifications that are acceptable to OSS communities. Resource usage is reduced using three techniques. First, the Linux kernel is tuned to reduce CPU overhead and memory usage. Second, unnecessary code is removed from userland libraries and commands. Third, security policy size is reduced by SEEdit. To be acceptable to OSS communities, the side effects of the code modifications were measured. In addition, the proposed system allows removed code to be selected by using a build flag at compilation time.

Embedded SELinux was evaluated on a SH-based evaluation board targeted for M2M gateways and CE devices. Benchmark results show that the SELinux overhead for read and write operations is almost negligible. SELinux's file space requirements are approximately 200 KB, and memory usage is approximately 500 KB, which consume approximately 1% of the flash ROM and RAM of the evaluation board, respectively. These results show that SELinux can be applied to embedded devices effectively. Regressions were also measured,

and no regression problems were observed. The modified codes were presented to OSS communities along with the evaluation results; as a result, the modifications were successfully merged into community codes.

# Chapter 4

# A Rule-based Sensor Data Aggregation Framework

## 4.1 Introduction

In this chapter, a rule-based sensor data aggregation system called the Complex Sensor Data Aggregator (CSDA) is proposed for the problem of developing and updating aggregation logics on M2M gateways. Firstly, issues in aggregation of sensor data for M2M gateways are described with categorization of aggregation process. Secondly, approach and design of CSDA is described. Finally, the effectiveness of CSDA is evaluated on an experimental environment.

## 4.2 Issues in aggregation of sensor data

In the following sections, after the functions in the process of sensor data aggregation for M2M gateways are subdivided into categories, the issues involved in writing the aggregation logic are described.

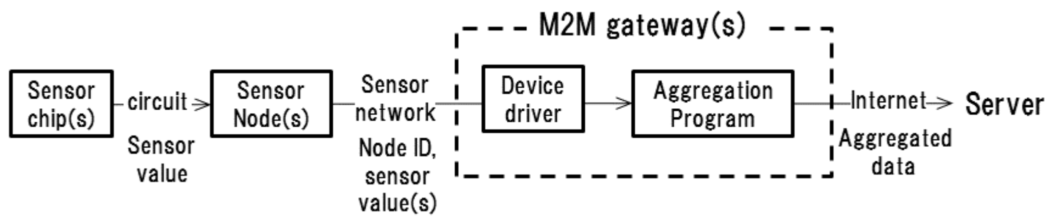


Figure 4.1 Overview of M2M system

### 4.2.1 Categories of functions in the process of sensor data aggregation on an M2M gateway

#### Process of sensor data aggregation

Before the functions in the sensor data aggregation process can be subdivided into categories, the process of inputting sensor data into the M2M gateway needs to be described. An overview of an M2M system is shown in Figure 4.1. Each sensor node in the sensor network is connected to sensor chips through a circuit. Sensor nodes are composed of a physical sensor network interface and a CPU, in which an embedded program fetches a value from the sensor chips and sends that value to the sensor network with the node's ID. The node ID identifies that node in the sensor network. The arbitration ID for the CAN protocol and the IEEE address for ZigBee are examples of node IDs. Note that multiple sensors may be connected to the same sensor node; in this case, multiple values are packed with that node's ID. The means by which fetched values are packed with node IDs, i.e., short or long, little-endian or big-endian, depend on the sensor node vendor. The M2M gateway and sensors are connected by various types of networks, and data are inputted into the gateway. For example, because the control of vehicles, industrial machinery, and medical equipment requires real-time communication, the CAN protocol, which has priority control features, is often used. When measuring temperature and humidity in a large area, such as a plant or building, the wireless protocol ZigBee is used in order to eliminate the cost of installing wires. M2M gateways have physical interfaces and device drivers in order to communicate with the sensor network protocol. They receive data frames, which include the previously described node IDs and sensor values.

The sensor data aggregation process runs on an M2M gateway, using node IDs and sensor values as input. The functions in the process can be subdivided into three categories: (1)

periodic statistical calculations to reduce data quantity, (2) filtering to reduce data frequency, and (3) concatenation to reduce communication overhead.

### Periodic statistical calculations

The quantity of data is reduced by calculating statistical values, such as averages and occurrence counts, from multiple data periodically within a given time frame. For example, temperature sensor data for the cooling of water and oil within construction machinery may be summarized by the sum, average, minimum, maximum, and value occurrence counts as taken at specified intervals (such as every 100 ms or every second). These processed data are then sent to a server [34]. Similarly, engine speed and temperature sensor data of farm machinery may be condensed to the average, minimum, and maximum and then sent to the server [36]. In addition, various statistic values of sensor data are utilized, such as standard deviation[97], moving average[98], Fourier transform[99] and so on. Furthermore, within a periodic process, statistical calculations can be performed multiple times, and multiple sensor data can be combined. In the evaluation of deviation from the stable state of construction machinery, variances are calculated for multiple temperature and pressure sensor values at given intervals, and then the variances are summed [35].

### Filtering

The quantity of data from an M2M gateway can be reduced by filtering out the low priority data. Filtering is categorized into two types: input filtering using node IDs and threshold filtering.

- Node ID filtering: To obtain important sensor values, the M2M gateway only takes the data frame, which includes the specific node ID. For example, since engine and transmission sensor values in construction machines are the ones used to detect failure, only these sensor values are gathered and sent to the server [34].
- Threshold filtering: The purpose of this filtering is to gather only the values that indicate trouble. For example, when a statistical value calculated from a construction machine's sensor data exceeds some threshold, the calculated value is sent to the server. Otherwise, no data are sent [35].

#### Concatenation

When data are sent to the server, protocol header information is attached. To reduce the overhead, multiple sensor data are joined and sent together. The timing of this depends on the type of data being collected and the reason for their collection. Referring again to the example of construction machinery, a summary of operation logs may be sent daily, and any engine sensor data indicating failure can be sent immediately [100].

However, additional statistical calculation combining results of previous periodic statistical calculations is not usually performed in M2M gateways because it consumes computing resource; this is normally the task of a server side application and is not within the scope of the aggregation process.

### 4.2.2 Issues with sensor data aggregation using firmware programming

The sensor data aggregation process previously described is programmed into the firmware of the M2M gateway. However, there are difficulties in developing and updating programs in firmware.

#### Developing firmware in resource-constrained environment

For industrial usage, cost and quality requirements for M2M gateways are strict because they often operate in extreme environments, sometimes operate in battery powered environment and the number of deployed devices is large. As a result, memory and CPU resources are much more constrained than on enterprise server. In addition, since product life cycle is longer than consumer devices, the growth of CPU and memory is slow. A CPU clock for an M2M gateway is often around 200-600 MHz, and it usually has about 1-64 MB of RAM. For example, RAM size for an M2M gateway to monitor construction machines [18] is only 2-8 MB, and field area network is 1Mbps CAN. CAN is fast and RAM resource becomes especially limited. Second example is an industrial controller with M2M gateway functionality [19] to monitor production lines. RAM size for user program is 8-16 MB and 10Mbps industrial Ethernet is used as a field area protocol. Another example can be found in general purpose industrial communication board, with a CPU clock of 250 MHz and 64

MB of RAM [20]. There are two issues with developing aggregation programs on firmware in a resource-constrained environment.

The first issue is the limitation of the SDK. Because it is difficult to use higher-level languages on this sort of hardware, the programming language is usually C. However, C is difficult to program, and the handling of memory is bothersome. Moreover, SDKs for M2M gateways vary depending on the gateway device. The development process typically consists of the following steps: writing aggregation logic in C on a PC, cross-compiling it for the gateway, producing a firmware image, and transferring it to the M2M gateway's flash memory. All of these processes are different depending on the M2M gateway because hardware, OS, and userland programs are not standardized.

The second issue is with managing limited memory, i.e., keeping within the limits specified by the M2M gateway's manufacturer and stable memory usage. In an M2M gateway, an out-of-memory condition can easily happen because the RAM size is often small and no swap space is prepared. If one application uses a lot of memory, other applications may not work because of the memory shortage. To avoid this problem, manufacturers of M2M gateways ask application developers to use a specified amount of memory and to test the memory usage of applications in advance. Therefore, memory usage of the aggregation logic must be kept within specified limits. In addition, the usage must also be static, i.e., the memory usage must be fixed at startup time and cannot change.

However, it is not easy to keep within a memory limitation because the data input rate is not stable. Buffer management for performing periodic statistical calculations is especially important. Data need to be stored in a buffer in order for statistical calculations to be performed for a specified time span. For example, when the average of temperature sensor data for 30 seconds is being calculated, all data generated from the temperature sensor during those 30 seconds are stored in buffer. The frequency of data generation varies depending on the configuration of the machine controlling the sensors. In sensors for a particular engine control unit, for example, the temperature sensor data are generated every second in its usual state, but when the engine begins to work hard, the control unit generates sensor data every 0.1 seconds. To process statistical calculation for sensor data arriving at such variable frequency, it would be easily handled if buffer were allocated dynamically, but memory usage needs to be static for M2M gateways. Therefore, buffers for statistical calculations must be allocated statically. Typically, static memory regions are managed by using a ring buffer

[75]. However, statistical information will be lost when input traffic increases. For example, an array of size 3,000 is allocated for a ring buffer to calculate the 30-second average for a sensor. When the frequency of input data is every 10 ms, the 3,000-element array is fully used and the 30-second average is calculated without a problem, but when the frequency increases and exceeds every 10 ms, statistical information will be lost. At a frequency of every 1 ms, only the last 3 seconds of data will be stored in the buffer, and the average for 30 seconds cannot be calculated. To prevent such a case, the size of the ring buffer needs to be large enough, but this would exceed the memory limit.

In the above example, note that all data do not necessarily have to be saved to calculate only average, because it can be obtained from the sum and the number of data. However, in order to calculate other statistic values such as moving average, deviation, Fourier transform and so on, all data within a specified time range have to be saved. In addition, in calculating only average it is preferable to save all data in a buffer to reduce context switch overhead between data receiving process and data calculation process.

#### Updating firmware

In order to modify the aggregation logic after the M2M gateways are deployed, the firmware needs to be updated. However, there are difficulties in modifying programs, installing firmware, and recovering from a failure.

- Modifying programs: The C program needs to be modified in order to change the aggregation logic; this involves difficulties similar to those previously described for the development of the C program. In addition, the C program requires repeat testing in order to avoid regression.

- Installing new firmware: After new firmware is developed, it must be installed on the devices. There are two ways of doing this. In the first method, field engineers perform the installation. Although this is the most reliable method, it is very expensive when the number of devices is 10,000 or 100,000 or more. To reduce this human cost, the second method is used, in which the updating is performed through the network. However, network costs are still an issue in this scenario. Since firmware is usually implemented on read-only file systems, entire file system images, which often exceed

10 MB, must be delivered via the mobile network, and the network cost over all the devices becomes impossible to ignore.

- Recovering from a failure: When the writing of firmware from the network fails because of a power outage during the update, the firmware is broken. To avoid this, a backup area, whose size is the same as the firmware's, is a necessary component of the flash ROM. In addition, the recovery procedure must be performed by an engineer.

## 4.3 Proposal of Complex Sensor Data Aggregator

### 4.3.1 Basic design of the CSDA

As described in the previous section, there are problems with programming and updating sensor data aggregation logic for M2M gateway firmware. Here, memory needs to be carefully managed. The memory usage of the aggregation logic must remain within the limit specified by the M2M gateway's manufacturer, and it should not increase after the aggregation logic begins to work.

In this section, a framework called CSDA is proposed as a solution to these problems. An overview of the CSDA is shown in Figure 4.2. The process of aggregation is specified in a configuration file, thus eliminating the need for programming in C. Typical logic components such as those for averaging and filtering are embedded in advance, and the combination of components to handle the required aggregation tasks is described in the configuration file. The process begins with the *process launcher*. It starts the *static aggregation processor* that is responsible for the aggregation process described in the configuration file. In order to prevent an increase in memory usage after startup, the static aggregation processor is composed of static working memory and a static number of threads. The threads handle the aggregation tasks by combining embedded logic components. In addition, to enable the adjusting of memory usage within the specified limits, the working memory size can be reduced by modifying the sampling configuration of a buffer, the details of which are described in section 4.3.2. The *update module* changes the configuration where aggregation logic is described, via the network. Here, it is assumed that the memory usage of the CSDA has been tested in a test environment and found to be within the limits before the configuration is changed. The

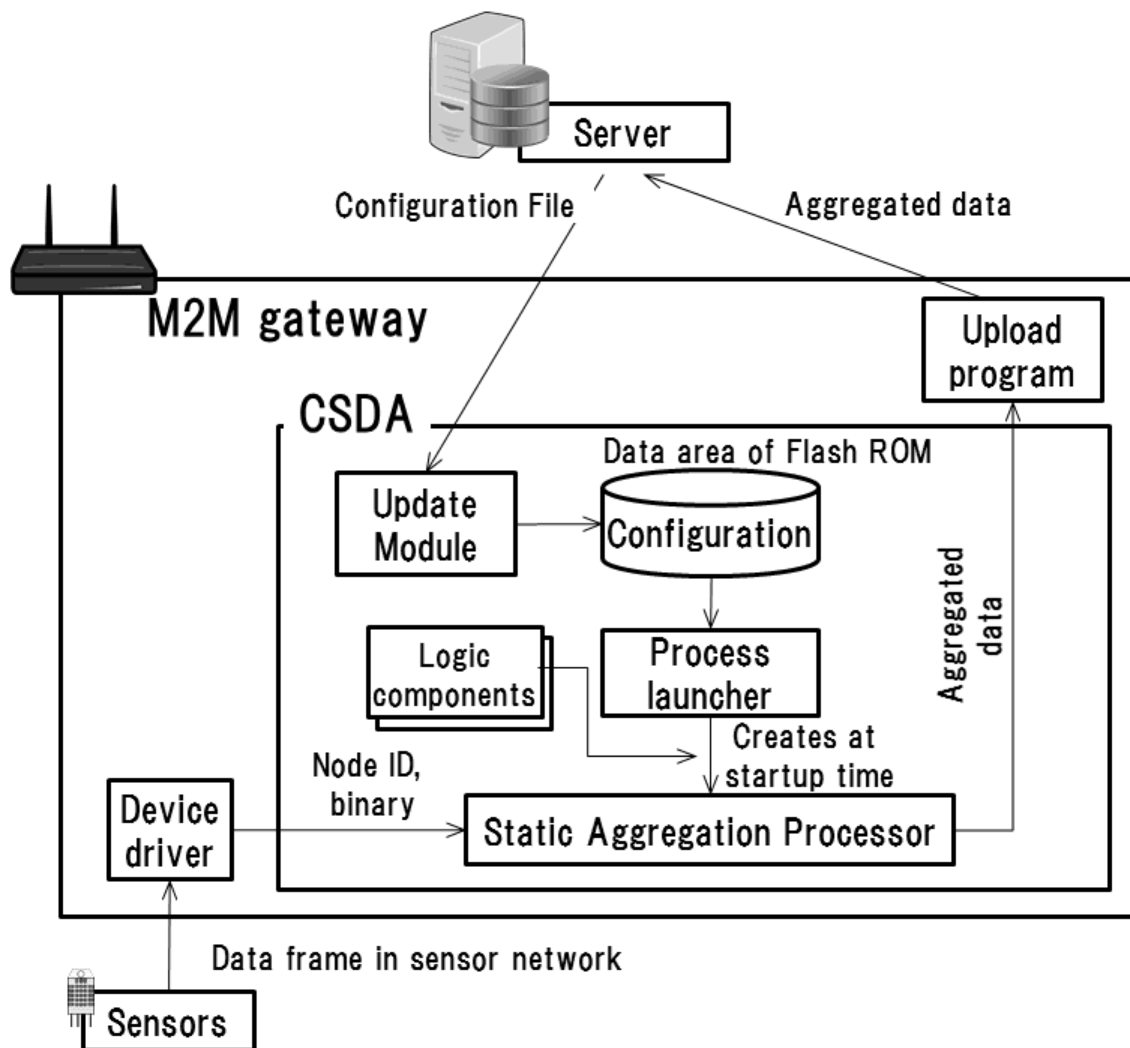


Figure 4.2 Architecture of the CSDA

detailed design of the static aggregation processor, configuration file, and update module are described in the following sections.

### 4.3.2 Design of static aggregation processor

Figure 4.3 shows the design of the static aggregation processor. Three types of threads aggregate sensor data according to a specified configuration. The number of threads is defined by the configuration and does not change after startup, thereby preventing an increase in memory usage. Buffers between threads, called the *sampling buffer* and the *processing cache*, are also designed to control memory usage; the sampling buffer in particular has the

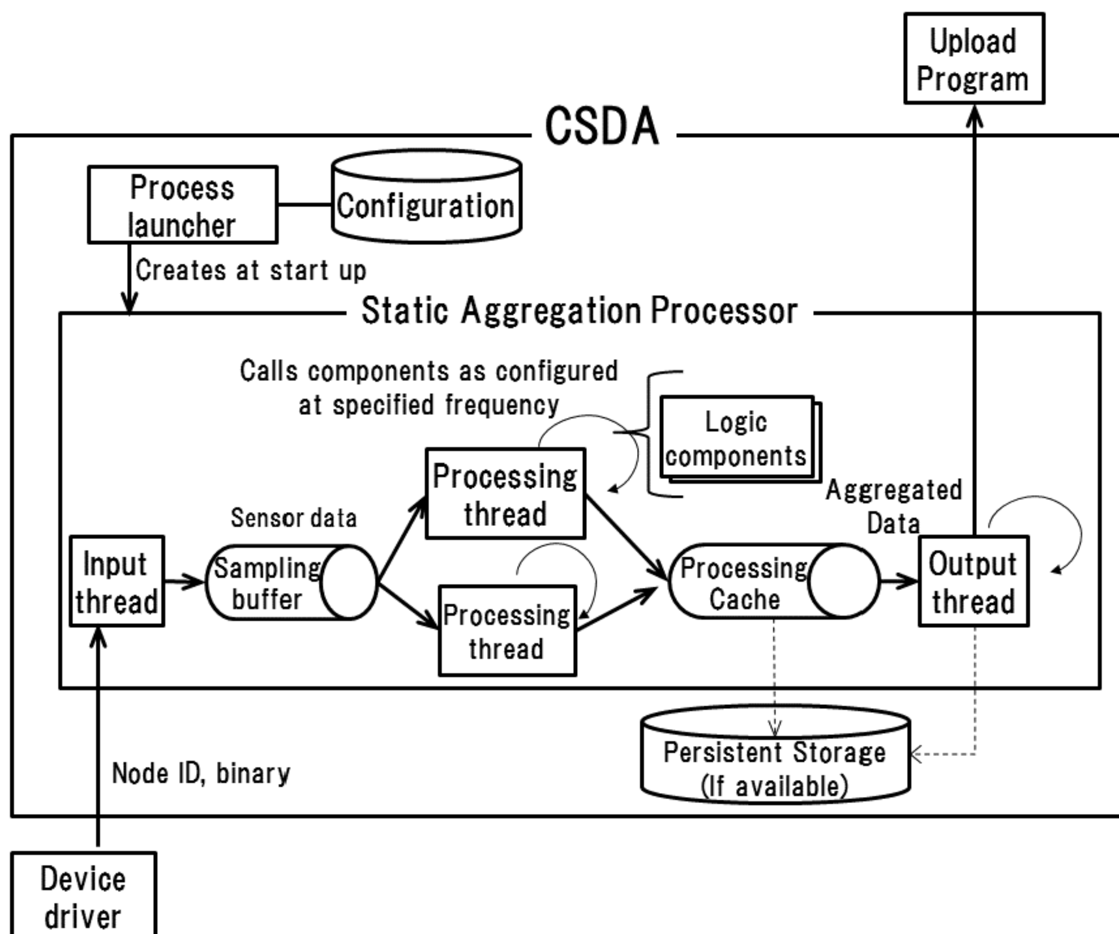


Figure 4.3 Design of Static Aggregation Processor

ability to adjust memory usage. The design of the threads and buffers is described next.

#### Threads for aggregation process

In order to cover the categorized functions of the aggregation process as described in section 4.2.1, the static aggregation processor handles sensor data in three steps: input, data processing, and output. These are implemented in the following three kinds of threads:

- Step1, input thread: One thread is launched at startup time. In this thread, node IDs and binary data, including sensor values, are passed from the device driver, and the node IDs are filtered out. Then, sensor values are extracted from the binary code and buffered in a memory area called the *sampling buffer*.

- Step2, data processing threads: This step takes data from the sampling buffer periodically at a configured rate (e.g., every 100 ms), and statistical calculations, such as the average, sum, minimum, maximum, and occurrence counts, are performed on the extracted sensor values. Threshold filtering is also performed on the sensor values and the calculated values. The two tasks, statistical calculation and threshold filtering, can be combined. The results of this data processing are saved in a memory area called the *processing cache*. The number of threads launched at startup depends on the variations of the frequency of data processing specified in the configuration file. For example, if data processing is configured for every 100 ms and every 10 ms, two threads are launched, and no new threads are launched after that.

- Step3, output thread: It reads the result data of the processing thread from the processing cache and concatenates them; then, the data are periodically passed to the data upload program, or passed immediately if the data indicate an event requiring urgent attention. In addition, if persistent storage is available, data can be stored there. The number of threads launched at startup also depends on the variations of the frequency of output processing specified in the configuration.

#### Memory management strategy

To ensure that memory usage remains within the CSDA's limit, the entire working memory size must be fixed at startup. However, the frequency of input data for processing threads and aggregated data from processing threads increases as sensor data traffic increases, and dynamic memory allocation can easily violate the memory usage limit as described in section 4.2.2. Therefore, for input and output of processing threads, static working memory area is prepared in order to establish the memory usage. The size of the static working area is determined from the CSDA's memory usage whose amount is specified by the manufacturer of the target M2M gateway. From the working area, a *sampling buffer* is allocated for input, and a *processing cache* is allocated for output at startup according to the configuration specified, and the allocated memory size does not change after that. The configuration needs to be adjusted not to consume the entire working memory area because startup will fail if the working memory area is fully used. The sampling buffer in particular is designed to adjust the memory usage because the quantity of sensor data input can become very large.

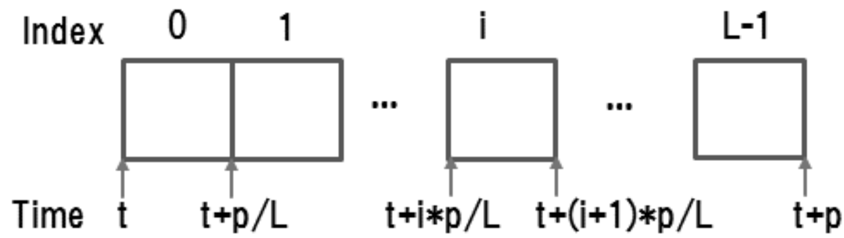


Figure 4.4 Array usage of sampling buffer

The design of these buffers is described next.

### Design of sampling buffer

As described in section 4.2.2, statistical information is lost when a simple ring buffer is used as the input buffer. To reduce such loss of statistical information and to enable memory usage adjustment, not all input data are saved in the sampling buffer when the frequency of data increases; instead, data are sampled at a configured rate. For example, when the sampling buffer is configured to save data every 100 ms and input data come every 10 ms, data are saved every 100 ms. This memory usage can be reduced by configuring the sampling rate because the amount of memory needed is smaller when the sampling rate is low.

Figure4.4 shows the details of the sampling buffer. At CSDA startup, the sampling buffer is allocated from the static working memory area for each input sensor from the working area; the array layout is as shown in the figure. Here, $p$ is the time period of the stored data and $L$ is the length of the array; their values are specified in the CSDA configuration file. At index $i$, representative data for the time between $t + (i - 1) * p/L$ and $t + i * p/L$ are stored. The representative data can be configured from the maximum value, minimum value, and latest value, and $p/L$ is the sampling rate. For convenience, the working of this array may be explained by an example. Assume that data are stored from sensor A for 30 seconds ($p = 30$), $L$ is configured as 300, and the sampling rate $p/L$ is 0.1 seconds. In array element 0, representative data between $t$ and $t + 0.1$ seconds are stored; in array element 1, representative data between $t + 0.1$ and $t + 0.2$ seconds are stored; and in array element $i$, representative data between $t + i * 0.1$ and $t + (i + 1) * 0.1$ seconds are stored. When the time reaches $t + 30$, data are stored from $i = 0$, and $t$ is incremented by 30. In this way, data from sensor A are stored every 0.1 seconds for 30 seconds. Then, in the data processing

step, data are taken from the buffer and used to perform the statistical calculations.

Developers of aggregation logic in CSDA can adjust $L$ according to the memory usage limits of the target M2M gateways. For example, assume that 2 bytes of sensor data are averaged for 20 seconds and that 200 KB of CSDA's working memory is available for the sampling buffer. In this case, the maximum value of $L$ will be 10,000, and the maximum sampling rate will be 0.1 ms. On the other hand, when only 2 KB of working memory is available for the sampling buffer, the maximum value for $L$ will be 100 and the sampling rate will be 10 ms.

#### Design of processing cache

This cache is used to store the result from the data processing step. There are two uses of this cache. The first one is simply to store the result of the data processing step and pass the result on to the next output step. The second use is to compute statistical values such as frequency counts over long time periods. For example, when a value is calculated every 100 ms and a frequency distribution for the value is created in the data processing step, in order to create a frequency distribution covering 1 minute, the frequency data must be stored in the processing cache.

The cache is also allocated from the static working memory area, and the allocation size is specified in the configuration file. When the cache becomes full, old cached data are removed or cached out to persistent storage if it exists. If data loss is not permitted, the cached-out data can also be sent to the server immediately.

### 4.3.3 Configuration language

A configuration language was designed to describe aggregation for the CSDA. XML was adopted as the text representation for the configuration language because it can be easily parsed and handled by other programs such as configuration tools and GUIs. However, it is difficult to handle an XML-based text configuration file on an M2M gateway because the XML parser usually consumes memory resource. To save memory, an XML-based configuration is encoded into a binary format outside the M2M gateway, and it is delivered and then loaded to the CSDA at startup. The configuration file for the CSDA consists of three

sections, corresponding to the steps in the aggregation process as described in section 4.3.2: input, data processing, and output.

#### Input section

In this section, the node ID filtering, target sensor values, and configuration of the sampling buffer are described. Figure4.5 shows an example of this configuration. Here, the *separatorId* attribute in line 1 refers to the node ID filtering. Only the input data whose node ID is described in separatorId are processed. In this example, only the data frames with node IDs of "1" are processed. The *offset* and *length* attributes correspond with the data extracted from the data frame payload. In line 2, 0 to 7 bits of payload are extracted and stored as a short data type (2 bytes) in a sampling buffer. Similarly, 8 to 15 bits of payload are extracted and stored in a sampling buffer. In order to identify the extracted data, the *rawDataId* attribute is used. Data extracted by line 2 are identified by "100", and data extracted by line 3 are identified by "101". Line 5 configures the sampling buffers. In this example, $p$ is configured as 4,000 ms, and $L$ is configured as 400. Therefore, the sampling rate (p/L) is 10 ms. As a result of the configuration, two sampling buffers with a length of 400, each of whose elements is 2 bytes (short) in size, are allocated from static working memory. This allocation is processed at CSDA startup, and if all working memory is used up, startup will fail with an error. If such a failure occurs, the *size* attribute will need to be reduced.

#### Data processing section

In this section, the configuration for periodic data processing, which consists of statistical calculations, threshold filtering, and a combination of the two, is specified. An example configuration is shown in Figure 4.6. The *sequencer* tag declares the start (line 1) and end (line 24) of the periodic data processing. A corresponding data processing thread is launched at startup. If there are two sequencer tag pairs, two threads are launched at startup. The *seqCacheSetting* tag in line 2 configures the processing cache. Here, 200 bytes are allocated from static working memory for the processing cache. The *runCondition* tag in lines 3-5 specifies the frequency of periodic processing. This configuration means that calculations within the sequencer tag (lines 1-24) are to be processed every 1,000 ms, and input data are

```
#Obtain data frame whose node ID is 1
1:<separator separatorId="1">
#Extract sensor data from data frame
#and identify them as 100,101
2:<offsetRawData rawDataId="100", dataType="short",
       length="8", offset="0"/>
3:<offsetRawData rawDataId="101", dataType="short",
       length="8", offset="8"/>
4:</separator>
#Configuration for sampling buffer: p is configured as 4000ms
#L is configured as 400
5:<rawCacheSetting size="400",period="4000",timeUnit="msec"/>
```

Figure 4.5 Part of configuration for input step

to be extracted from the sampling buffer for the last 1,000 ms . After the runCondition tag, processes are described in *step* tags, and each step tag has a step number specified in the stepId attribute.

A statistical calculation is specified by an *operation* tag within the step tag. Embedded logic components are called as identified by the *method* attribute. Lines 6-10 are an example of a statistical calculation. Here, the method attribute is “average”, and the averaging calculation is called from embedded logic components. In order to refer to data from the input step, a *rawRefId* in an *arg* tag is used. Data corresponding to rawDataID = “100” are taken from the sampling buffer for the last 1,000 ms and are averaged. Lines 11-14 are an example of threshold filtering. If the result of the previous step (average) is greater than 10, then go to the next step; if it is not greater than 10, then go to step 4 (line 23) as specified by the *false* attribute. In the next steps (lines 15-22), a histogram of frequency counts is taken for data whose rawDataID = “101”. Results of this calculation are also identified by a *stepId* attribute in the *step* tag, to be used by others. For example, the result of the histogram step is identified as “3”. Processing cache usage is configured by the *cache* attribute in the step tag. If it is “yes”, as in line 15, the result of the calculation is saved in the processing cache.

```
 1:<sequencer sequencerId="1">
#Processing cache size is 200 bytes
 2:    <seqCacheSetting size="200"/>
#Line 6-24 are processed every 1000 ms
 3:<runCondition runType="periodic">
 4:   <term timeUnit="msec">1000</term>
 5: </runCondition>
# Average is taken for data identified as 100
 6:<step stepId="1">
 7:  <operation method="average">
 8:    <arg key="refRawId">100</arg>
 9:  </operation>
10:</step>
# If above average exceeds threshold, goto line 15
# Else go to line 23
11:<step stepId="2">
12:  <filter operator="greater equal" rightOperand="10"/>
13:   <nextstep false="4"/>
14:</step>
# Histogram is taken for data identified as 101
15:<step stepId="3" cache="yes">
16:  <operation method="histogram">
17:    <arg key="series">20</arg>
18:    <arg key="min">0</arg>
19:    <arg key="max">100</arg>
20:    <arg key="refRawId">101</arg>
21:  </operation>
22:</step>
# Some  calculations...
23:<step stepId="4" cache="yes">
    (snip)
24: </sequencer>
```

Figure 4.6 Part of the configuration for data processing step

### Output section

The process of concatenating and sending data to the server is specified. An example configuration for the output step is shown in Figure 4.7. The *deliverer* tag declares output

```
1:<deliverer delivererId="101">
# Data is outputted every 60 min
2: <runCondition runType="periodic">
3:   <term timeUnit="min">60</term>
4: </runCondition>
# Data created in data processing
#(line 15-22,23-24 in previous figure) are concatenated
5: <dataFormat>
6:   <stepId>3</stepId>
7:   <stepId>4</stepId>
8: </dataFormat>
9:</deliverer>
```

Figure 4.7 Part of the configuration for output step

section. A corresponding output processing thread is launched at startup. If there are two deliverer tag pairs, two threads are launched at startup. In the example, only one thread is launched. In lines 2-8, every 60 minutes, data that are identified as “3” and “4” are taken from the processing cache and concatenated. Finally, the result is passed to another program to send the data to the M2M server.

### 4.3.4 Update module

This module downloads the configuration from the network and rewrites it. The configuration is written in data area in Flash ROM or RAM which is usually used to store various configuration data. Failure recovery must be considered similar to firmware update. To achieve this, aggregation processors and update modules run as separate processes. The update is performed as follows:

- Step 1: The update module downloads the configuration file onto the RAM, and a copy of the old configuration file is created in the flash ROM as a backup.
- Step 2: The update module writes a new configuration to the flash ROM.
- Step 3: The update module restarts the aggregation processor
- Step 4: The aggregation processor loads new configuration from the flash ROM

Even if writing the configuration fails in Step 2 because of an unexpected power outage, the update can be resumed from Step 1, and the old configuration can be used, regardless of whether the resuming download fails. The flash ROM size required for the backup is at most the size of the configuration.

## 4.4 Evaluation

CSDA was implemented in C, and the CSDA effectiveness was evaluated in an experimental environment similar to an M2M gateway. CSDA was compared with the development of dedicated logic in a traditional C program. First, to investigate the advantage of CSDA, the developmental process and the updating of aggregation logic on an M2M gateway was evaluated. In addition, the capability of the sampling buffer to adjust memory usage within the limits was also evaluated. Second, overhead was evaluated. CSDA has performance overhead compared with a dedicated C program because CSDA includes logic to process various calculations according to configurations. To verify that the level of overhead is acceptable, memory and CPU usage were measured and compared with those of a dedicated C program.

### 4.4.1 Experimental setup

CPU clocks for M2M gateways are typically around 200-600 MHz, and RAM sizes are around 1-64 MB, as was stated in section 4.2.2. The Armadillo 420 [101] was used as an evaluation device because its CPU power and RAM size are similar to the above specifications and its SDK is available to the public. Its specifications are shown in Table 4.1. The CAN was selected as a sensor network because it is widely used in industrial devices. CSDA was implemented in C and ported to the device in the following evaluations.

### 4.4.2 Evaluation of advantage of CSDA

To verify the advantage of CSDA over a dedicated C program, the development and update processes are compared. Next, the capability of the sampling buffer to adjust working memory usage within the set limits is evaluated.

Table 4.1 Embedded system used in the evaluation

| | Specification |
|---|---|
| CPU | Freescale i.MX25(ARM926EJ-S) 400 MHz |
| RAM | LPDDR SDRAM 64 MB |
| Sensor network interface | ISO11898 compliant CAN interface |
| OS | Linux 2.6.26 |

**Development of the aggregation process**

The productivity of C and of CSDA in the development of an aggregation process are compared. The following aggregation process was used for the evaluation:

- Input process step: Take two kinds of sensor value for Sensor A and Sensor B.
- Data processing step: When the 30-second average value for Sensor A is greater than a given threshold, create a histogram of the Sensor B values.
- Output step: Send the histogram data once each day.

The number of steps was calculated for the dedicated C code and the number of tag pairs was counted for the CSDA configuration. The number of steps required in the C program was 575, and the number of tag pairs in the CSDA configuration was 29. The semantics of C and XML are different, but the amount of configuration is much less than in the C code. The sampling buffer was not implemented in the dedicated C program; if it were to be implemented, the amount of the program would increase more.

In addition, there are the following three difficulties in writing C code. The first is that of bit operation. In the input step, sensor data are extracted from the binary frame. In order to handle binary communication, bit operations are used, as shown in Figure4.8; these operations are difficult to read. Eight variables and three arrays are used in a mere three lines. Using the CSDA, on the other hand, data extraction can be described in one simple line. The second difficulty is with memory operations. As is well known, failure to handle C arrays causes many problems. If there is a bug in the boundary handling, the memory may be destroyed, and this weakness can be also exploited in a buffer overflow attack [22]. The

```
* Part of C code
for(idx2 = 0; idx2 < byteLength ;idx2++){
  if(offsetRem > rightShift){
       output[idx2] = (data[idx3]>>rightShift)
       & forAnd[leftShift];
* CSDA configuration
 <extract rawId="10" offset="0" len="8"/>
```

Figure 4.8 Comparison of C and CSDA at input step

CSDA hides such memory operations. Third is thread management. The thread needs to be separated for the input process and data processing in order to enable the receiving of data during the data processing. Thread programming depends on the OS, and programmers need to learn manners. Moreover, the shared resource needs to be handled carefully because failure can lead to resource destruction or a deadlock. Here as well, the CSDA hides the operations.

### Updating aggregation logic

Updating of the aggregation logic in CSDA is compared with traditional firmware updating through consideration of the following aspects during the update process:

(1) Delivery of update data: In a firmware update, the entire image needs to be delivered. In some cases its size can exceed 10 MB. In CSDA, the binary configuration file needs to be delivered, but its size is small. For example, the size of the binary configuration used in the previous section is 1 KB. This will save network bandwidth and loading of the delivery server.

(2) Logic updating: Since the size of the binary configuration file is much smaller than a firmware image, the time for rewriting the configuration file will be also much smaller than that needed for rewriting a firmware image.

(3) Failure recovery: In most M2M gateways, there is a failure recovery feature for firmware updates. If a firmware update fails, it is recovered from backup firmware. However,

intervention by an engineer in the field is necessary since the M2M gateway is not functioning after the failure.

In CSDA, there is a failure recovery feature as described in section 4.3.4. Recovery after a failure can be performed without the need for an engineer in the field because other parts of the M2M gateway and update module are working. For example, if there is a monitor function in the update module that watches the CSDA status, the monitor function can detect a failure and recover the configuration file from backup.

**Capability of sampling buffer to reduce memory usage**

To verify that memory usage can be reduced by adjusting the sampling buffer, the CSDA memory usage was measured for varying array lengths of the sampling buffer. In the evaluation, it was assumed that data are inputted from three sensors whose data are 2 bytes in size and that three sampling buffers are prepared to store data from them for 30 seconds. The CSDA was simply configured to perform 30-second averaging of the sensor data stored in the sampling buffers.

Memory usage of the CSDA was measured as follows. As described in section 4.3.2, CSDA has static working memory for the sampling buffer. Logic to measure the size of CSDA's unused working memory ($A$) was inserted in the CSDA, and the total CSDA memory usage ($B$) was measured by free command. (Note that $B$ is the difference in the output of the free command before and after starting CSDA.) Here, the actual CSDA memory usage is $B$ - $A$. $A$ and $B$ were measured when the length of array for the sampling buffer was 300 (for a sampling rate of 100 ms), 3,000 (for a sampling rate of 10 ms), and 30,000 (for a sampling rate of 1 ms).

The measured memory usage for the sampling buffer is shown in Table 4.2. L is length of array for sampling buffer and R is sampling rate. $B$ - $A$ is displayed as memory usage there. For example, in the case of $L = 30{,}000$, $A$ was 132 KB and B 580 KB; thus, the CSDA memory usage was 580 - 132 = 448 KB. From the table, it is observed that CSDA memory usage decreases as $L$ decreases and that the rate of decrease is just 7 bytes. It is confirmed that the CSDA memory usage can be adjusted in a memory-constrained environment by configuring $L$. For example, assuming only 300 KB is allowed by the M2M gateway's manufacturer, when $L = 30{,}000$ ($R = 1$ ms), the memory usage will be 448 KB, exceeding the limits. In such a case, memory usage can be reduced by reducing $L$. The value of $L$ that achieves the best

Table 4.2 Memory usage of CSDA varying configuration of sampling buffer

| | L=30,000 R=1 ms | L=3,000 R=10 ms | L=300 R=100 ms |
|---|---|---|---|
| Memory usage (KB) | 448 | 259 | 240 |

sampling rate within the memory usage limit can also be easily calculated. Since the rate of memory usage decrease is 7 bytes, by reducing $L$ by 21,143 (= 148 KB/7), memory usage will be just 300 KB. The resulting $L$ is 8,857 (= 30,000 - 21,143), and the sampling rate will be 3.4 ms (= 30 s / 8,857).

### 4.4.3 Evaluation of overhead

When aggregation logic is implemented on the CSDA, memory usage is expected to be higher than that with a dedicated C program, because the CSDA is implemented in C and logic to handle various aggregations is included. CPU usage is expected to be higher as well. To verify that such overheads are acceptable, the RAM and CPU usage are compared with those in a dedicated C program.

#### Overhead in RAM usage

The overhead in memory usage was measured as follows:

- Aggregation logic that simply performs 30-second averages of data from three sensors and discards the outputs was implemented both in a dedicated C program and in CSDA. Neither the sampling buffer nor the processing cache were implemented in the dedicated C program; it reads 30 seconds of data into a dynamically allocated buffer.
- To determine the increase in CSDA memory usage for handling various aggregations, $C$ and $D$ were measured as follows:
  $C$: (memory usage of CSDA) - (size of sampling buffer)
  $D$: (memory usage of dedicated C program) - (size of dynamically allocated buffer)
  $C$ - $D$ thus represents the CSDA overhead.

The result of measuring the CSDA overhead ($C$ - $D$) was 72 KB, where $C$ was 248 KB and $D$ was 176 KB. This increase is not significant, because even if the RAM size is 8 MB, it is less than 1%.

#### Overhead of CPU usage

In order to see the CPU overhead, complicated aggregation logic was configured and evaluated on the device, the same as that of Table 4.1. The logic for the evaluation was a statistical calculation to predict faults in a construction machine using data from three sensors. The calculation is composed of 30 calculation elements [35]. To simulate heavy traffic, sensor data were inputted every 1 ms, then the statistical calculation was performed every 100 ms. CPU time was measured for 1 second. CPU time was also measured for a dedicated C program where only the aggregation logic was coded.

The observed CPU usage of the CSDA was 1.5% (CPU time 15 ms), and that for the dedicated C program was less than 0.1% (CPU time less than 1 ms). The CPU usage of the CSDA was not significant even under heavy traffic with complicated aggregation logic, but was worse than for the C program. The performance loss is due to overhead in parsing the calculation rule. When a use case with more complicated aggregation logic is encountered in the future, it will need to be tuned.

## 4.5 Conclusions

To reduce the server load and communication cost of M2M systems, sensor data are aggregated in an M2M gateway. The aggregation logic is usually programmed in C rather than higher-level programming languages because the CPU and memory resources are constrained. However, there are difficulties with programming in C and with updating the programs. A framework called complex sensor data aggregator (CSDA) was proposed to solve such difficulties. The proposed CSDA is highlighted as follows.

- CSDA enables sensor data aggregation in M2M gateways without the need for programming. This CSDA supports categorized data aggregation methods in three steps: the input, the periodic data processing, and the output steps. In each step, behavior is configured using XML-based rules.

- In order to keep CSDA within the memory limit specified by the M2M gateway's manufacturer, the number of threads and the size of working memory is static after startup, and the size of the working memory can be adjusted by configuration of a sampling setting for a buffer for sensor data input.

- Experimental results on an evaluation board show that developing CSDA configurations is much easier than programming in C. The amount of configuration is less than 10% of the comparable C code. Memory usage can also be reduced by adjusting the sampling setting. In addition, the basic memory usage and CPU usage of the CSDA are not significant for M2M gateways, i.e., the CPU usage with the CSDA is about 1.5% for complicated aggregation logic, and the increase in memory usage compared with dedicated C logic is about 70 KB.

# Chapter 5

# Conclusions

## 5.1 Concluding remarks

In this dissertation, a study of improvement of security and data aggregation for M2M gateway systems was described, and following results were obtained.

In the chapter 1, functions of M2M gateways in IoT services were described, and problems were introduced. As an increase of IoT service opportunities and number of devices connected to the Internet, the importance of security and data aggregation for M2M gateways will also be significant. In developing security and data aggregation functions, three constraints specific to M2M gateways must be considered, i.e. hardware resource limitation, difficulty in updating programs, and various development environments. However, existing security and data aggregation technologies are not enough considering these constraints. From security perspective, SELinux is an effective protection method because it works without updates on various system configurations and resource usage does not increase after deployment. Two problems to apply SELinux to M2M gateways were described, i.e. creation of security policy and basic resource usage. From data aggregation perspective, aggregation logics on M2M gateways are effective to reduce data sent to the Internet, because software management in each sensor nodes is not necessary. Problems of describing and updating aggregation logics were introduced. Development of aggregation logics requires knowledge specific to M2M gateways and consideration of resource limitation issues. In addition, modification of aggregation logics requires firmware updates, which are risky.

In the chapter 2, a SELinux security policy configuration system called SEEdit was proposed. SEEdit facilitates creating security policies by utilizing a higher level language called

SPDL and SPDL tools. SPDL reduces the number of permissions by integrated permissions and removes label configurations. SPDL tools generate security policy configurations from access logs and tool user's knowledge about applications. The proposed system was evaluated on an experimental environment targeted for M2M gateways. Experimental results prove that SEEdit can create practical security policies because it makes describing configurations a semi-automated process, creates security policies containing less than 500 lines of configuration and less than 500 KB of memory footprint. As a result of this research, SEEdit was provided as an OSS [102], and actually used in a major commercial embedded Linux distribution [103].

In the chapter 3, Embedded SELinux was proposed to make resource usage of SELinux acceptable for embedded systems such as M2M gateways. In Embedded SELinux, resource usage is reduced by employing three techniques. First, the Linux kernel is tuned to reduce CPU overhead and memory usage. Second, unnecessary codes are removed from userland. Third, security policy size is reduced with SEEdit. To facilitate acceptance by OSS communities, build flags are used to bypass modified code, such that it will not affect existing features; moreover, side effects of the modified code were carefully measured. Embedded SELinux was evaluated using an evaluation board targeted for M2M gateways, and results show that its read/write overhead is almost negligible. SELinux's file space requirements are approximately 200 KB, and memory usage is approximately 500 KB; these account for approximately 1% of the evaluation board's respective flash ROM and RAM capacity. Moreover, the modifications did not result in any adverse side effects. As a result of the research, modified codes were successfully merged into communities' codes and create basics of subsequent projects such as SE Android [66] and a major commercial embedded Linux distribution.

In the chapter 4, a data aggregation framework called CSDA was proposed to support developing and updating aggregation logics on M2M gateways. The functions comprising the data aggregation process are subdivided into the categories of filtering, statistical calculation, and concatenation. The proposed CSDA supports this aggregation process in three steps: the input, periodic data processing, and output steps. The behaviors of these steps are configured by an XML-based rule. The rule is stored in the data area of flash ROM or RAM and is updatable through the Internet without the need for a firmware update. In addition, in order to keep within the memory limit specified by the M2M gateway's manufacturer, the

number of threads and the size of the working memory are static after startup, and the size of the working memory can be adjusted by configuring the sampling setting of a buffer for sensor data input. The proposed system was evaluated in an M2M gateway experimental environment. Results show that developing CSDA configurations is much easier than using C because the amount of configuration is less than 10% of the comparable C code. In addition, the performance evaluation demonstrates the proposed system's ability to operate on M2M gateways. Consequently, a prototype developed for the study creates the basis of a commercial product called the Entier Stream Data Aggregator [104] and is being used in actual M2M gateways.

## 5.2 Future directions

Finally, future problems and directions of this research are described.

(1) A SELinux configuration tool with higher level language
There are three remaining problems in SEEdit. First is trade-offs where generated security policies may have security problems as described in chapter 2. For that issue, a tool that evaluates generated configurations would be necessary. Second issue is co-existing with refpolicy. To make wider application of SEEdit, it is preferable that generated policies can be appended to refpolicy. However, SEEdit cannot be used with refpolicy because type configurations generated by SPDL converter conflict with existing type configurations in refpolicy. The SPDL converter therefore should be improved in order to resolve such conflicts. Third issue is application of SEEdit to cloud systems. For example, SELinux has been extended to cloud systems such as Docker [105]. However, they assume standard refpolicy and do not compatible with policies generated by SEEdit. SEEdit or cloud systems have to be extended.

(2) Tuning SELinux with contribution to OSS communities
Embedded SELinux focuses on reducing resource usage, but does not solve userland permission issues. Permissions are also defined in userland, e.g. restarting applications in a framework. To enable checking such permissions in SELinux, SELinux exposes API. For example, SE Android brings the SELinux check to Android applications by inserting SELinux API into Android's framework. However, Android can only accommodate mobile devices with GUI and is not suitable for M2M gateways. There are

frameworks for devices without GUI, such as OSGi [70] and Allseen [106]. Integrating SELinux to these frameworks will be required in the future.

Security policies were created by SEEdit, but there are redundant configurations and the size can be reduced more. Yagi et.al [56] proposed a system to remove unnecessary rules from SELinux security policy based on access log. The system may be used to reduce the size of security policies created by SEEdit.

(3) A rule-based sensor data aggregation framework

The limitation of the CSDA is that it cannot describe statistical calculations whose logics and configuration elements are not already embedded. In order to use the CSDA in a variety of cases, it is assumed that enough sets of statistical calculations are supported. In practice, "enough sets" are prepared after the CSDA is used in multiple situations. However, if "enough sets" are defined, the memory consumption may become too large because of the amount of embedded logics. To solve this problem, a plug-in framework would be useful. If a new calculation logic is necessary, it could then be supported by adding plug-ins. Memory usage would thereby be saved since only the necessary plug-ins would need to be installed in the CSDA.

# Acknowledgements

I am cordially grateful to Associate Professor Toshihiro Yamauchi of the Graduate School of Natural Science and Technology at Okayama University for his visionary directions and constructive suggestions on this research activity and on writing this dissertation.

I would like to thank to Professors Masanobu Abe and Akira Nagoya of the Graduate School of Natural Science and Technology at Okayama University for their numerous suggestions for revising this dissertation.

I would like to also express my gratitude to Dr. Jonathan Stanton of Assistant Professor of the George Washington University (Currently, Spread Concepts LLC) for his supervision and valuable suggestions during my study at the M.S. course of the George Washington University.

I would like to express my appreciation to Dr. Yoshiki Sameshima of Hitachi Solutions, Ltd. (Currently retired), and Dr. Takashi Onoyama of Hitachi Solutions, Ltd. for their support and useful advice ever since I joined Hitachi Solutions, Ltd. I would like to also give my thanks to my seniors, colleagues, and juniors of the company and Hitachi, Ltd.

I would also like to acknowledge people in OSS communities: SELinux mailing list, BusyBox mailing list, CELinux Forum (Currently Linux Foundation) and Japan SELinux Users Group, for valuable suggestions about the study.

Finally I would like to show my deepest gratitude to my wife and sons.

# References

[1] Gartner, Inc: Predicts 2015: The Internet of Things, https://www.gartner.com/doc/2952822/predicts--internet-things (accessed 2016-05-28).

[2] Apache Hadoop: http://hadoop.apache.org/ (accessed 2016-07-03).

[3] Apache Spark: http://spark.apache.org/ (accessed 2016-07-03).

[4] Komatsu: Komtrax, http://www.komatsuamerica.com/komtrax (accessed 2016-05-28).

[5] Hitachi Construction Machinery: Global-eService, http://www.hitachi-c-m.com/global/businesses/products/global_e-service.html (accessed 2016-05-28).

[6] Yanmar: Yanmar's Advanced Technology Gives Customers a SmartAssist, http://yanmar.com/news/contents/105278.php (accessed 2016-05-28).

[7] Federal Ministry for Economic Affairs and Energy of Germany: Platform Industrie 4.0, http://www.plattform-i40.de/ (accessed 2016-05-28).

[8] Industrial Internet Consortium: http://www.iiconsortium.org/ (accessed 2016-05-28).

[9] Lowe's Companies, Inc.: IRIS home automation system, https://www.irisbylowes.com/ (accessed 2016-05-28).

[10] Control 4 corporation: http://www.control4.com/ (accessed 2016-05-28).

[11] ZigBee Alliance: Interconnecting ZigBee & M2M Networks, http://docbox.etsi.org/workshop/2011/201110_m2mworkshop/03_m2mcooperation/zigbee_taylor.pdf (accessed 2016-05-28).

[12] CiA: Controller Area Network, `http://www.can-cia.org/` (accessed 2016-05-28).

[13] Rojas, C. and Morell, P.: Guidelines for Industrial Ethernet infrastructure implementation: A control engineer's guide, *In Proc. of 52nd IEEE-IAS/PCA Cement Industry Technical Conference*, pp. 1–18 (2010).

[14] The European Telecommunications Standards Institute: Machine-to-Machine communications(M2M) functional architecture, ETSI TS 102 690 V1.1.1 (2011).

[15] Grau, A.: How to Build a Safer Internet of Things, IEEE Spectrum. `http://spectrum.ieee.org/telecom/security/how-to-build-a-safer-internet-of-things` (accessed 2016-05-28).

[16] Malik, M.: Meet Remaiten a Linux bot on steroids targeting routers and potentially other IoT devices, ESET. `http://www.welivesecurity.com/2016/03/30/meet-remaiten-a-linux-bot-on-steroids-targeting-routers-and-potentially-other-iot-devices/` (accessed 2016-05-28).

[17] EMC Corporation: Digital Universe Invaded By Sensors, `http://www.emc.com/about/news/press/2014/20140409-01.htm` (accessed 2016-05-28).

[18] Quake Global: Q4000, `http://quakeglobal.com/files/tinymce/uploaded/documents/Q4000%202013.pdf` (accessed 2016-05-28).

[19] Hitachi Industrial Equipment Systems: HX series, `http://www.hitachi.com/New/cnews/month/2015/11/151116.html` (accessed 2016-05-28).

[20] Hitachi Super LSI Systems: Communication module for industrial devices, `http://www.hitachi-ul.co.jp/system/cmodule/index.html` (accessed 2016-05-28).

[21] Murdock, R. and Rommel, C.: *The Global Market for IoT and Embedded Operating systems*, VDC Research Group, Inc (2015).

[22] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., A. Grier, A., Wagle, P., Zhange, Q. and Hinton, H.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks, *In Proc. of the 7th USENIX Security Symposium*, pp. 63–78 (1998).

[23] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N. and Boneh, D.: On the effectiveness of address-space randomization, *In Proc. of the 11th ACM conference on Computer and communications security*, pp. 298–307 (2004).

[24] Bojinov, H., Boneh, D., Cannings, R. and Andmalchev, I.: Address space randomization for mobile devices, *In Proc. of the Fourth ACM Conference on Wireless network security*, pp. 127–138 (2011).

[25] Schwartz, E., Avgerinos, T. and Brumley, D.: Q: Exploit Hardening Made Easy, *In Proc. of the 20th USENIX Security Symposium*, pp. 379–394 (2011).

[26] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S. and Walter, T.: Breaking the memory secrecy assumption, *In Proc. of the Second European Workshop on System Security*, pp. 1–8 (2009).

[27] Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S. and Farrell, J.: The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments,, *In Proc. of 21st National Information Systems Security Conference*, pp. 303–314 (1998).

[28] Wright, C., Cowan, C., Smalley, S., Morris, J. and Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux Kernel, *In Proc. of 11th USENIX Security Symposium*, pp. 17–31 (2002).

[29] Loscocco, P. and Smalley, S.: Integrating Flexible Support for Security Policies into the Linux Operating System, *In Proc. of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, pp. 29–42 (2001).

[30] Harada, T., Handa, T., Hashimoto, M. and Tanaka, H.: Mandatory Access Control Method Based on Application Execution State, *IPSJ Journal*, Vol. 53, No. 9, pp. 1–18 (2012).

[31] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P. and Gligor, V.: SubDomain:Parsimonious Server Security, *In Proc. of the 14th USENIX conference on System Administration*, pp. 355–368 (2000).

[32] Walsh, D.: SELinux Constrains Samba Vulnerability, `http://danwalsh.livejournal.com/10131.html` (accessed 2016-05-28).

[33] Saini, M. and Kaur, K.: Review of Open Source Software Development Life Cycle Models, *International Journal of Software Engineering and Its Applications*, Vol. 8, No. 3, pp. 417–434 (2014).

[34] Murakami, T., Saigo, T., Ohkura, Y., Okawa, Y. and Taninaga, T.: Development of Vehicle Health Monitoring System(VHMS/WebCARE) for Large-Sized Construction Machine, *Komatsu Tech Rep*, Vol. 48, No. 150, pp. 15–21 (2003).

[35] Fujiwara, J. and Suzuki, H.: Device for collection construction machine operation data, WIPO Patent, WO2013077309 A1 (2013).

[36] Shinohara, Y. and Sakamoto, H.: Remote monitoring terminal device for traveling work machine or ship, WIPO patent, WO2013080712 A1 (2013).

[37] Bandyopadhyay, S. and Bhattacharyya, A.: Lightweight Internet protocols for web enablement of sensors using constrained gateway devices, *In Proc. of 2013 International Conference on Computing, Networking and Communications(ICNC2013)*, pp. 334–340 (2013).

[38] Stanford-Clark, A. and Truong, H. L.: MQTT For Sensor Networks (MQTT-SN) Protocol Specification, `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf` (accessed 2016-05-28).

[39] Shuang, K., Shan, X., Sheng, Z. and Zhu, C.: An Efficient ZigBee-WebSocket Based M2M Environmental Monitoring System, *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pp. 322–326 (2014).

[40] Kitagami, S., Kaneko, Y. and Suganuma, T.: Method of Autonomic Load Balancing for Long Polling in M2M Service System, *In Proc. of Advanced Information Networking and Applications Workshops*, pp. 294–299 (2012).

[41] EMC Education Services: *Data Science and Big Data Analytics: Discovering, Analyzing, Visualizing and Presenting Data*, John Wiley & Sons, Inc. (2015).

[42] Blais, B.: Technology Strategies for Big Data Analytics, SAS Presentation delivered at the Big Analytics Roadshow (2012). `http://www.slideshare.net/AsterData/sas-ny-big-analytics-conference` (accessed 2016-05-28).

[43] SELinux Example Policy Objectives: https://www.nsa.gov/what-we-do/research/selinux/policy-objectives.shtml (accessed 2016-05-28).

[44] SELinux Reference Policy: https://github.com/TresysTechnology/refpolicy/wiki (accessed 2016-05-28).

[45] SETools: https://github.com/TresysTechnology/setools3 (accessed 2016-05-28).

[46] Marouf, S. and Shehab, M.: SEGrapher: Visualization-based SELinux policy analysis, *In Proc. of 4th Symposium on Configuration Analytics and Automation*, pp. 1–8 (2011).

[47] Jaeger, T., Edwards, A. and Zhang, X.: Managing access control policies using access control spaces, *In Proc. of the seventh ACM symposium on Access control models and technologies (SACMAT 02)*, pp. 3–12 (2002).

[48] Guttman, J., Herzog, A., Ramsdell, J. and Skorupka, C.: Verifying information goals in security-enhanced linux, *Journal of Computer Security*, Vol. 13, No. 1, pp. 115–134 (2005).

[49] Amthor, P., Kuhnhauser, W. and Polck, A.: Model-based safety analysis of SELinux security policies, *In Proc. of 5th International Conference on Network and System Security*, pp. 208–215 (2011).

[50] MacMillan, K.: Core Policy Management Infrastructure for SELinux, Presentation at SELinux Symposium 2005 (2005). http://selinuxsymposium.org/2005/presentations/session3/3-2-macmillan.pdf (accessed 2016-05-28).

[51] Lugo, P. C., Flores, J. J. and Garcia, J. M.: An Architecture for Systematic Administration of SELinux Policies in Distributed Environments, *International Journal of Computers and Communications*, Vol. 1, No. 4, pp. 127–135 (2007).

[52] Honda, A., Asakura, Y., Saida, Y. and Watanabe, M.: Policy addition mechanism of secure OS for embedded system, *IPSJ SIG Technical Report*, Vol. 2008, No. 21, pp. 109–114 (2008).

[53] Linux manpages for audit2allow(1): http://linuxcommand.org/man_pages/audit2allow1.html (accessed 2016-05-28).

[54] Denis, J.: Setroubleshoot: A User Friendly Tool to Diagnose AVC Denials, *In Proc. of 2007 Security Enhanced Linux Symposium* (2007). http://selinuxsymposium.org/2007/papers/09-setroubleshoot.pdf (accessed 2016-06-25).

[55] Sniffen, B., Ramsdell, J. and Harris, D.: Guided Policy Generation for Application Authors, *In Proc. of 2006 Security Enhanced Linux Symposium* (2006). http://selinuxsymposium.org/2006/papers/14-guided-polgen.pdf (accessed 2016-06-25).

[56] Yagi, S., Nakamura, Y. and Yamauchi, T.: Proposal of a Method to Automatically Reduce Redundant Security Policy of SELinux, *IPSJ Journal*, Vol. 5, No. 2, pp. 63–73 (2011).

[57] SLIDE: http://oss.tresys.com/projects/slide (accessed 2016-05-28).

[58] CDS Framework IDE: http://oss.tresys.com/projects/cdsframework (accessed 2016-05-28).

[59] Ueno, S., Mizukami, T., Takeuchi, H., Ogawa, T., Aoki, J., Shirai, A., Imachi, R., Yokogami, H., Nakamura, Y. and Shimura, T.: Development of aiding programs for Security-Enhanced Linux, http://www.ipa.go.jp/security/fy15/development/selaid/documents/selaid-abst.pdf (accessed 2016-05-28) (2004).

[60] Kuliniewicz, P.: SENG: An Enhanced Policy Language for SELinux, *In Proc. of 2006 Security Enhanced Linux Symposium* (2006). http://selinuxsymposium.org/2006/papers/09-SENG.pdf (accessed 2016-06-25).

[61] Sellers, C., Athey, J., Shimko, S., Mayer, F. and MacMillan, K.: Experiences Implementing a Higher-Level Policy Language for SELinux, *In Proc. of 2006 Security Enhanced Linux Symposium* (2006). http://selinuxsymposium.org/2006/papers/08-higher-level-experience.pdf (accessed 2016-06-25).

[62] Coker, R.: Porting NSA Security Enhanced Linux to Hand-held devices, *In Proc. of of 2003 Ottawa Linux Symposium*, pp. 117–127 (2003).

[63] Vogel, B. and Steinke, B.: Using SELinux Security Enforcement in Linux-based Embedded Devices, *In Proc. of the 1st International Conference on Mobile Wireless MiddleWare, Operating Systems and Applications*, pp. 15:1–15:5 (2007).

[64] Fiorin, L., Ferrante, A., Padarnitsas, K. and Carucci, S.: Hardware-assisted security enhanced Linux in embedded systems, *In Proceedings of the 5th Workshop on Embedded Systems Security*, No. 3, pp. 3:1–3:7 (2010).

[65] Shabtai, A., Fledel, Y. and Elovici, Y.: Securing Android-Powered Mobile Devices Using SELinux, *IEEE Security and Privacy Magazine May-June 2010*, pp. 36–44 (2010).

[66] Smalley, S. and Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android, *In Proc. of 20th Network and Distributed System Security Symposium*, pp. 20–38 (2013).

[67] Muthukumaran, D., Schiffman, J., Hassan, M., Sawani, A., Rao, V. and Jaeger, T.: Protecting the integrity of trusted applications in mobile phone systems, *Security and Communication Networks*, Vol. 4, No. 6, pp. 633–650 (2011).

[68] Zhang, X., Aciicmez, O. and Seifert, J.: Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms, *In Proc. of the 1st International Conference of Security and Privacy in Mobile Information and Communication Systems*, pp. 71–82 (2009).

[69] Nahari, H.: Trusted Secure Embedded Linux: From Hardware Root of Trust to Mandatory Access Control, *In Proc. of 2007 Ottawa Linux Symposium*, pp. 79–86 (2007).

[70] OSGi Alliance: *OSGi Service Platform, Release 3*, IOS Press, Inc (2003).

[71] Li, Y., Wang, F., He, F. and Li, Z.: OSGi-based service gateway architecture for intelligent automobiles, *In Proc. of Intelligent Vehicles Symposium 2005*, pp. 861–865 (2005).

[72] Kura: `https://eclipse.org/kura/` (accessed 2016-05-28).

[73] Li, Z., Zhao, D., Wen, Y. and Mu, Y.: Design of an OSGi-Based WSN Gateway, *In Proc. of IEEE Ninth International Conference on Mobile Ad-hoc and Sensor Networks*, pp. 395–398 (2013).

[74] The STREAM Group: STREAM: The Stanford Stream Data Manager, *IEEE Data Engineering Bulletin*, Vol. 26, No. 1, pp. 19–26 (2003).

[75] Katsunuma, S., Honda, S., Sato, K. and Tanaka, H.: The Static Scheduling Method in Data Stream Management for Automotive Embedded Systems, *IPSJ Journal Database*, Vol. 5, No. 3, pp. 36–50 (2012).

[76] Yamamoto, M. and Koizumi, H.: An Experimental Evaluation of Distributed Data Stream Processing using Lightweight RDBMS SQLite, *IEEJ Transactions on Electronics, Information and Systems*, Vol. 133, No. 11, pp. 2125–2132 (2013).

[77] Madden, S., Franklin, M., Hellerstein, J. and Hong, W.: TinyDB: An Acquisitional Query Processing System for Sensor Networks, *ACM Transactions on Database Systems*, Vol. 30, No. 1, pp. 122–173 (2005).

[78] Boulis, A., Han, C., R. Shea, R. and Srivastava, M.: SensorWare: Programming sensor network beyond code update and querying, *Pervasive and Mobile Computing*, Vol. 3, No. 4, pp. 386–412 (2007).

[79] Muro, K., Urano, T., Odaka, T. and Suzuki, K.: AirsenseWare: Sensor-Network Middleware for Information Sharing, *In Proc. of 3rd International Conference on Intelligent Sensors, Sensor Networks and Information 2007(ISSNIP2007)*, pp. 497–502 (2007).

[80] Yamaguchi, H., Hiromori, A., Uchiyama, A., Higashino, T., Yanagiya, N., Nakatani, T., Tachibana, A. and Hasegawa, T.: Multi-dimensional sensor data aggregator for adaptive network management in M2M communications, *In Proc. of 2015 IFIP/IEEE International Symposium on Integrated Network Management*, pp. 1047–1052 (2015).

[81] Papageorgiou, A., Cheng, B. and Kovacs, E.: Real-time data reduction at the network edge of Internet-of-Things systems, *In Proc. of 11th International Conference on Network and Service Management*, pp. 284–291 (2015).

[82] Matamoros, J. and Anton-Haro, C.: Data aggregation schemes for Machine-to-Machine gateways: Interplay with MAC protocols, *In Proc. of Future Network and Mobile Summit*, pp. 1–8 (2012).

[83] Papageorgiou, A., Schmidt, M., Song, J. and Kami, N.: Smart M2M Data Filtering Using Domain-Specific Thresholds in Domain-Agnostic Platforms, *In Proc. of 2013 IEEE International Congress on Big Data*, pp. 286–293 (2013).

[84] Boebert, W. and Kain, R.: A Practical Alternative to Hierarchical Integrity Policies, *In Proc of the Eighth National Computer Security Conference*, pp. 18–27 (1985).

[85] Smalley, S., Vance, C. and Salamon, W.: Implementing SELinux as a Linux Security Module, Technical Report 01-043, NAI Labs (2006).

[86] Smalley, S.: Configuring the SELinux policy, Technical Report 02-007, NAI Labs (2002).

[87] GNU m4: `http://www.gnu.org/software/m4/m4.html` (accessed 2016-05-28).

[88] system-config selinux: `http://man7.org/linux/man-pages/man8/system-config-selinux.8.html` (accessed 2016-05-28).

[89] Yamaguchi, T., Nakamura, Y. and Tabata, T.: Integrated Access Permission: Secure and Simple Policy Description by Integration of File Access Vector Permission, *Proc. The 2nd International Conference on Information Security and Assurance(ISA2008)*, pp. 40–45 (2008).

[90] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. and Lepreau, J.: The Flask Security Architecture: System Support for Diverse Security Policies, *In Proc. of the 8th Conference on USENIX Security Symposium*, pp. 123–139 (1999).

[91] McVoy, L. and Staelin, C.: lmbench:Portable tools for performance analysis, *In Proc. of USENIX 1996 Annual Technical Conference*, pp. 279–295 (1996).

[92] UnixBench: `https://github.com/kdlucas/byte-unixbench` (accessed 2016-05-28).

[93] BusyBox: `http://www.busybox.net` (accessed 2016-05-28).

[94] Walsh, D.: SELinux Targeted vs Strict Policy History and Strategy, Presentation at SELinux Symposium 2005 (2005). `http://selinuxsymposium.org/2005/presentations/session4/4-1-walsh.pdf` (accessed 2016-05-28).

[95] Yamamoto, K. and Yamauchi, T.: Evaluation of Performance of Secure OS Using Performance Evaluation Mechanism of LSMPMON, *IPSJ Journal*, Vol. 52, No. 9, pp. 2596–2601 (2011).

[96] GNU Diffutils: `http://www.gnu.org/software/diffutils/` (accessed 2016-05-28).

[97] Genterberg, E., Ghasemzadeh, H., Jafari, R. and Bajcsy, R.: A Segmentation Technique Based on Standard Deviation in Body Sensor Networks, *In Proc. of 2007 IEEE Dallas Engineering in Medicine and Biology Workshop*, pp. 63–66 (2007).

[98] Zhuang, Y., Chen, L., Wang, X. and Lian, J.: A Weighted Moving Average-based Approach for Cleaning Sensor Data, *In Proc. of 27th International Conference on Distributed Computing Systems*, p. 38 (2007).

[99] Canli, T., Gupta, A. and Khokar, A.: Power Efficient Algorithms for Fast Fourier Transform over Wireless Sensor Networks, *In Proc. of IEEE Computer Systems and Applications*, pp. 549–556 (2006).

[100] Takishita, T., Murakami, K., Seki, K. and Morishita, K.: Application of ICT to Lifecycle Support for Construction Machinery, *Hitachi Review*, Vol. 62, No. 2, pp. 107–112 (2013).

[101] Atmark Techno, Inc.: Armadillo-420, `http://armadillo.atmark-techno.com/armadillo-420` (accessed 2016-05-28).

[102] SELinux Policy Editor Project: `http://seedit.sourceforge.net` (accessed 2016-05-28).

[103] Wind River Systems, Inc: What's New in Wind River Linux 4 Updata Pack2, `http://www.windriver.com/products/linux/Whats_New_LE_4_Update_Pk2_0811.pdf` (accessed 2016-05-28).

[104] Hitahci Solutions, Ltd: Entier Stream Data Aggregator, `http://www.hitachi-solutions.co.jp/entiersda/` (accessed 2016-05-28).

[105] Docker Docs: Docker Security, `https://docs.docker.com/engine/security/security/` (accessed 2016-05-28).

[106] Allseen alliance: `https://allseenalliance.org/` (accessed 2016-05-28).