Future Generation Computer Systems 76 (2017) 350–357

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Design and implementation of the secure compiler and virtual machine for developing secure IoT services

YangSun Lee[a], Junho Jeong[b], Yunsik Son[b,c,*]

[a] Department of Computer Engineering, Seokyeong University, 16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, Republic of Korea
[b] Department of Computer Engineering, Dongguk University, 26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, Republic of Korea
[c] Department of Brain and Cognitive Engineering, Korea University, 145 Anam-ro, Seongbuk-ku, Seoul 136-713, Republic of Korea

HIGHLIGHTS

- Secure software for developing secure/trustworthy services for IoT was proposed.
- A secure compiler was used in the development phase to eliminate the weaknesses.
- A virtual machine was used in the operating phase to watch the abnormal behaviors.

ARTICLE INFO

Article history:
Received 15 October 2015
Received in revised form
24 February 2016
Accepted 23 March 2016
Available online 8 April 2016

Keywords:
Secure software
IoT services
S/W weakness
Program analysis
Compiler construction
Virtual machine

ABSTRACT

Recent years have seen the development of computing environments for IoT (Internet of Things) services, which exchange large amounts of information using various heterogeneous devices that are always connected to networks. Since the data communication and services occur on a variety of devices, which not only include traditional computing environments and mobile devices such as smartphones, but also household appliances, embedded devices, and sensor nodes, the security requirements are becoming increasingly important at this point in time. Already, in the case of mobile applications, security has emerged as a new issue, as the dissemination and use of mobile applications have been rapidly expanding. This software, including IoT services and mobile applications, is continuously exposed to malicious attacks by hackers, because it exchanges data in the open Internet environment. The security weaknesses of this software are the direct cause of software breaches causing serious economic loss. In recent years, the awareness that developing secure software is intrinsically the most effective way to eliminate the software vulnerability, rather than strengthening the security system of the external environment, has increased. Therefore, methodology based on the use of secure coding rules and checking tools is attracting attention to prevent software breaches in the coding stage to eliminate the above vulnerabilities. This paper proposes a compiler and a virtual machine with secure software concepts for developing secure and trustworthy services for IoT environments. By using a compiler and virtual machine, we approach the problem in two stages: a prevention stage, in which the secure compiler removes the security weaknesses from the source code during the application development phase, and a monitoring stage, in which the secure virtual machine monitors abnormal behavior such as buffer overflow attacks or untrusted input data handling while applications are running.

1. Introduction

Recently, the expansion of computing environments to the IoT (Internet of Things), and mobile and cloud computing have resulted in privacy and system security issues becoming more important. Especially, the software included in mobile applications will always be vulnerable to possible malicious attacks by hackers, because it exchanges data in the Internet environment. These security weaknesses are the direct cause of software breaches, thereby causing serious economic loss. Moreover, in recent years the computing environment has been changing into a complicated system composed of various and heterogeneous sensors, IoT/embedded devices, mobile devices, PCs, and servers from the traditional environments.

* Corresponding author at: Department of Brain and Cognitive Engineering, Korea University, 145 Anam-ro, Seongbuk-ku, Seoul 136-713, Republic of Korea.
E-mail addresses: yslee@skuniv.ac.kr (Y. Lee), yanyenli@dongguk.edu (J. Jeong), sonbug@dongguk.edu, sonbug@korea.ac.kr (Y. Son).

http://dx.doi.org/10.1016/j.future.2016.03.014
0167-739X/© 2016 Elsevier B.V. All rights reserved.

In this environment everything is connected; hence, it is difficult to apply conventional application development methods and execution environments to this complicated system. Thus, IoT services are vulnerable to serious security problems such as hacking and exploiting, because almost all the devices of IoT systems are connected to the Internet and transmit data over the network. IoT sensors or devices are more exposed to relatively serious security threats compared to a traditional server system inside a firewall or IDS (Intrusion Detection System). When such terminal devices are under external attack, the entire IoT-based services are unable to operate normally because of the abnormal behavior.

In this regard, offering a secure coding guide or static analysis tools to solve software weaknesses from the coding stage is a trend, nowadays. If weaknesses are considered and prevented from the software development stage, enormous cost can be cut, compared to the efforts to recognize and correct weaknesses in the operation stage, and also huge contribution can be made to the development of safe software from hackers [1,2].

Our research team is working to solve this problem with the aim of producing high-quality/trustworthy IoT services by developing technology for IoT secure software development and execution based on a compiler and virtual machine. In this paper, we propose the use of a secure compiler and virtual machine with a stack monitoring method to develop secure IoT applications and protect abnormal behavior in computing environments containing various IoT devices.

A secure compiler was designed for preventing software weaknesses in the source code during the application development phase, and it is combined with a traditional compiler and weakness analyzer to generate the target code and remove the weaknesses. The secure compiler is implemented in conjunction with a virtual machine, which monitors abnormal behavior such as buffer overflow attacks or untrusted input data handling to protect the system while the applications are running.

The contents of this paper are as follows. First, in Section 2, secure coding, weakness analysis tools, and a smart cross platform are examined. Next, in Section 3, the technique proposed in this paper is introduced. In Section 4, the results obtained by applying the proposed method are analyzed and evaluated. Lastly, in Section 5, the conclusion and future direction of research are discussed.

2. Related studies

2.1. Secure coding

The software of today exchanges data in the Internet environment, thereby making it difficult to secure the validity of the data input and output. The possibility of being maliciously attacked by unknown and random invaders exists. This weakness has been the direct cause of software security incidents, which generate significant economic losses or social problems [1].

Security systems, installed to prevent security incidents from occurring, mostly consist of firewalls, user authentication systems, etc. However, according to a Gartner report [2], 75% of software security incidents occur because of weaknesses in the application programs. Therefore, rather than strengthening the security systems for the external environment, the creation of more secure software code by programmers is a more fundamental and effective method of increasing the security levels. However, efforts to reduce the weaknesses of a computer system are still mainly biased to network servers.

Recently, there has been recognition of this problem and therefore research on secure coding, that is, writing secure codes from the development stage [3,4] onwards, is being carried out actively. Especially, CWE (Common Weakness Enumeration) [5], an organization that analyzes the weaknesses that can arise from programming language, has analyzed and specified the various weaknesses that can occur in the source code creation stage of different languages. Also, CERT (Computer Emergency Response Team) [6] has defined secure coding rules to ensure secure source code creation. In Cigital [7], the weaknesses can be eliminated by using the 61 rules classified according to the Seven Pernicious Kingdoms [8] classification method proposed by Katrina Tsipenyuk, Brian Chess, and Gary McGraw. The coding rule suggested by Cigital is defined in XML form and can be used as an input in weakness analyzers and other programs. Industries prone to fatal mistakes due to software defects, such as the airplane and car industry, have implemented coding rules, such as JSF and MISRA Coding Rule [9], to contribute towards high quality software development.

2.2. Source code weakness analyzer

According to a report by Gartner [2], 75% of recent software security incidents were caused by applications containing vulnerable points; thus, the effective detection and elimination of possible weaknesses in a program from the application development stage has become a very important issue.

The source code weakness analyzer is a tool which has been developed to automatically examine the weaknesses within source code after it has been created by a programmer. Programmers aspire to have weaknesses within their programs to be entirely eliminated. However, it is difficult to acquire expert knowledge about weaknesses and it is difficult to recognize how to alter such weaknesses. Therefore, there is a need for a tool capable of automatically analyzing weaknesses at the source code level. There exists a suitable weakness analysis method depending on each weakness and these are broadly classified into static and dynamic analysis methods. The static method uses technology that does not require the subject program to run and uses methods such as token, AST (Abstract Syntax Tree), CGF (Control Flow Graph), DFG (Data Flow Graph). The dynamic method uses technology that performs a level-by-level analysis of programs while they are running and it uses certain codes that can either be used during execution time or by library mapping to carry out the analysis.

MOPS (MOdel Checking Programs for Security properties) [10] is a model testing machine developed at the University of California, Berkeley. MOPS defines the properties of security weakness factors, and has been standardized using limited automata. Accordingly, weaknesses that have been modeled can all be examined at low analysis costs. However, since it does not analyze the flow of data, there is a limit to the weaknesses that can be analyzed. Safe-Secure C/C++ by Plum Hall [11] is a type of compiler that has combined a compiler with a software analysis tool. Safe-Secure C/C++ only focuses on eliminating buffer overflow. Execution programs created using this software are capable of eliminating buffer over-flows 100% and have less than a 5% decrease in function compared to execution files created by ordinary compilers. Coverity's Coverity SAVE [12], is a static analysis tool for source codes. Coverity SAVE shows all weaknesses discovered in codes as a list. Each list includes details on the location of and reason for weaknesses discovered within each list. Fortify Static Code Analyzer (SCA) [13] is a weakness detection tool. Fortify SCA supports C/C++, Java, and other languages, and uses both static and dynamic analysis to detect weaknesses in source codes. The detected weaknesses are given to the user along with statistical data. Compass [14] is an open source static analysis tool for C/C++ based on ROSE [15]. Rule-based Compass uses the source-to-source framework ROSE for source code transformations, allowing users to modify the domain-specific rules sets of this tool. Sparrow [16] is a tool that carries out

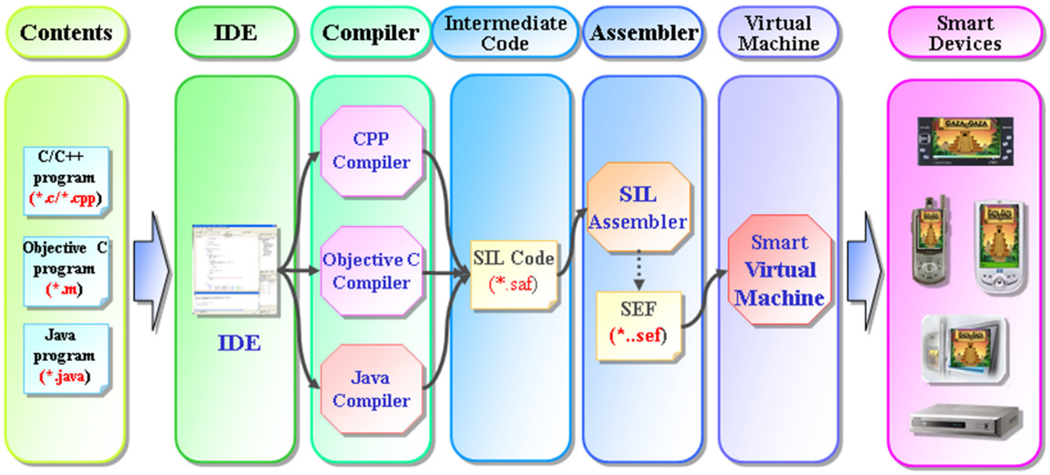

Fig. 1. System configuration of the Smart Cross Platform.

a semantic analysis to detect buffer overruns, memory leakage, and other critical memory errors and is an automatic program error analyzer based on semantic analysis. It provides information on the analysis time, error path, and memory status of the analyzed errors.

2.3. *Smart Cross Platform*

Existing smart phone content development environments require different object codes to be presented for each target device or platform. The languages that can be developed also vary depending on the platforms. The Smart Cross Platform [17] was developed to support platform-independent downloading and executing application programs in the various smart devices. In addition, the Smart Cross Platform supports multiple programming languages by using the intermediate language named SIL, which is designed to cover both procedural and object-oriented programming languages. Currently, the platform supports C/C++, Objective C, and Java, which are the languages most widely used by developers.

The Smart Cross Platform consists of three main parts: a compiler, assembler, and virtual machine. It is designed as a hierarchal structure to minimize the burden of the retargeting process. Fig. 1 shows a model of the Smart Cross Platform.

The SIL code is a result of the compilation process and is changed into smart executable format (SEF) through an assembler. The smart virtual machine (SVM) then runs the program after receiving the SEF. The SVM is composed of five major modules: the SEF loader, stack-based interpreter, SVM built-in libraries, native interfaces, runtime environments, and runtime environments consisting of an exception handler, memory management, and a thread scheduler [18]. The SVM is designed to easily add debugging interfaces, profiling interfaces, etc. The system configuration of the SVM is shown in Fig. 2.

The secure compiler and virtual machine with stack monitoring proposed in this paper are extensions of the compiler and virtual machine of the Smart Cross Platform.

3. Secure compiler and virtual machine for IoT services

3.1. *System model*

In current heterogeneous computing environments, such as the IoT, in which everything is connected, it is difficult to apply conventional application development methods and execution environments. Our research team is working on solving this problem with the aim of producing high-quality/trustworthy IoT services. Our technology provides three major features: (i) the same developmental environment and common runtime, (ii) software weakness elimination methods and secure runtime monitoring module, (iii) performance enhancement of the low computing power for IoT devices using cloud services and offloading techniques with a virtual machine. Fig. 3 shows the complete system model consisting of secure compilers and a virtual machine.

In this paper, as a first step of this research effort, we propose a secure coding rule-based compiler and virtual machine with secure runtime focused on stack monitoring.

3.2. *Secure compiler*

The secure compiler was designed by adding a secure coding rule checker and a static weakness analyzer to the compiler model of the Smart Cross Platform [19,20]. In this study, the secure compiler comprises eight parts as can be seen in Fig. 4.

The secure compiler provides secure features to prevent software weaknesses in the input program source code in C/C++. It was designed with six general compiler parts – scanner (lexical analysis), parser (syntax analysis), SDT (syntax directed translation), semantic analyzer, ICG (intermediate code generator), and optimizer – and two kinds of secure parts: a secure coding rule checker and a static weakness analyzer. The detailed information for each part is as follows.

The scanner, parser, and SDT modules can easily be grouped as a processor to analyze the input C/C++ programs and generate an analyzed AST for the input programs.

The semantic analyzer checks the process of collecting symbol information on the AST level, to verify cases which are grammatically correct but semantically incorrect. Moreover, it uses the AST and symbol table to carry out a semantic analysis of statements and creates a semantic tree as a result. A semantic tree is a data structure to which semantic information is added from an AST and is not only used for generating the VM (virtual machine) code but also for analyzing software weaknesses.

The code generation module receives the semantic tree as an input after the analysis is complete and generates a VM code which is semantically equal to the input program in C/C++.

The secure coding rule checker is the module that detects the rule violations of the input programs. The coding rules are defined by meta-language that was designed to describe the secure policy of the target programming languages. The defined rules are interpreted by the rule checker, which analyzes the violations using the input semantic tree with interpreted rule information [21].

The static weakness analysis module analyzes the control flow and data flow of a source program by using the symbol information

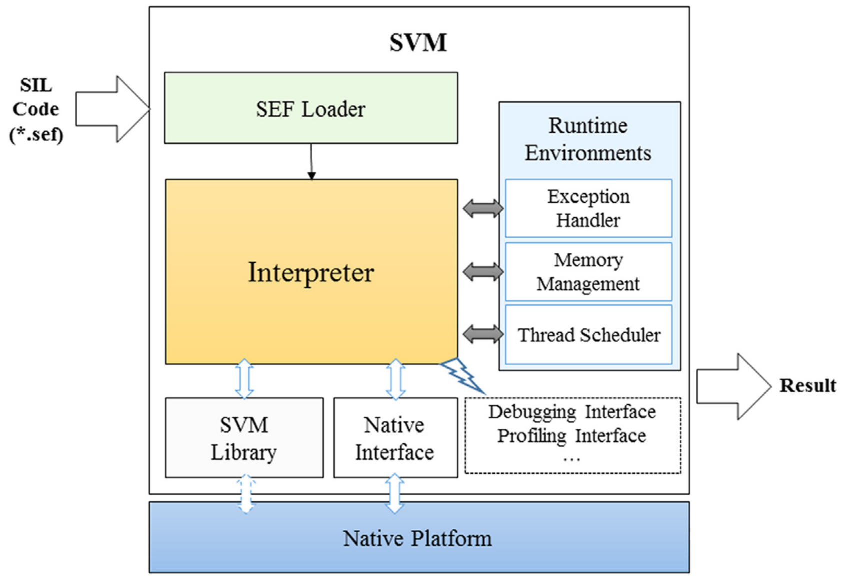

Fig. 2. System configuration of the smart virtual machine.

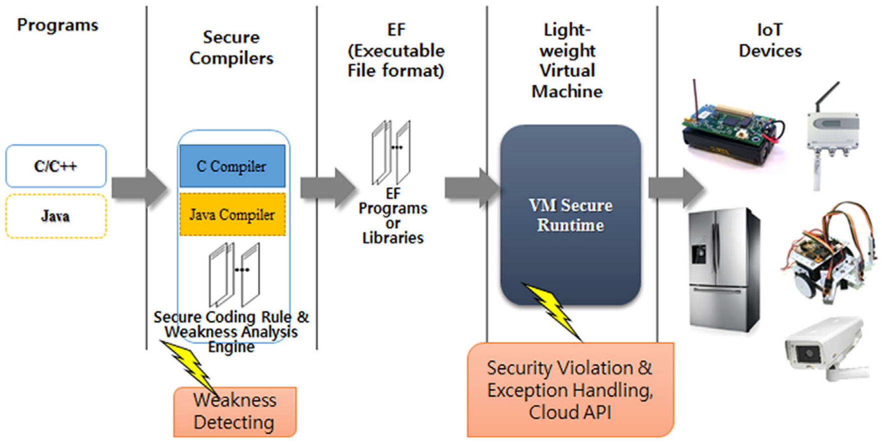
Programs
Secure Compilers
EF (Executable File format)
Light-weight Virtual Machine
IoT Devices
C/C++
Java
C Compiler
Java Compiler
Secure Coding Rule & Weakness Analysis Engine
EF Programs or Libraries
VM Secure Runtime
Weakness Detecting
Security Violation & Exception Handling, Cloud API

Fig. 3. Secure development and execution model for IoT services.

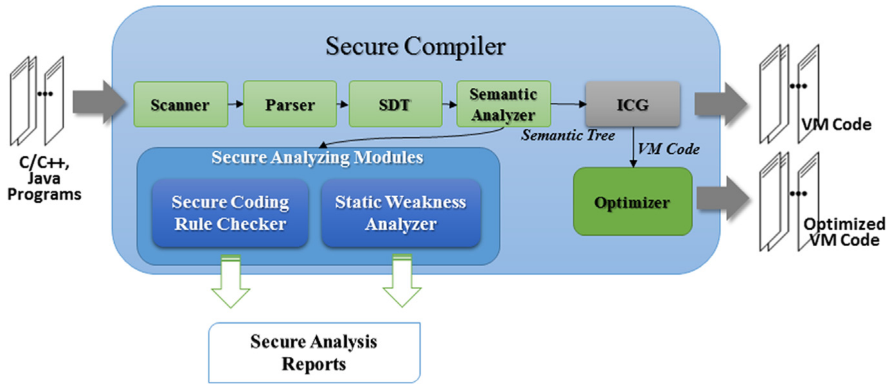

Fig. 4. Proposed secure compiler model for C/C++ languages.

and semantic tree generated by the front end of the compiler. Some weaknesses are too complicated to allow precise assessment by the rule checker. In that case, the rule checker generates too many false alarms when targeting weaknesses. Weaknesses such as these require the use of a static weakness analysis module with 1:1 mapping routines for specific weakness analysis.

The software weaknesses that are analyzed by the proposed compiler are collected and categorized from the top-level weaknesses defined by CWE and OWASP (Open Web Application Security Project) [22] for embedded systems, networks, the IoT, and C/C++ programming languages.

The secure coding rules for IoT services are defined and categorized by IoT and mobile application-specific weakness groups defined during previous research [21,23]. The major weakness rules are listed in Table 1.

Table 1
Secure coding rules for IoT applications.

Category	Weakness type (No. sub category)	Major secure coding rule
Language independent	Code Quality(3)	Do not reuse public identifiers
	Input validation(1)	Sanitize untrusted data passed across a trust boundary.
	Security features(1)	Do not allow privileged blocks to leak sensitive information across a trust boundary
	Class(3)	Defensively copy private mutable class members before returning their references
	Time and states(2)	Avoid deadlock by requesting and releasing locks in the same order
	Error handling(2)	Prevent exceptions while logging data
Language dependent	Libraries(4)	Do not use vulnerable APIs
		Do not use deprecated methods
	Security features(4)	Do not use vulnerable Algorithms
		Do not leak personal information
	Resource usage(7)	Do not allow unprivileged resources
	System events(2)	Must write the event handlers
	Runtime environments(1)	Consider multiple vendor's device characteristics

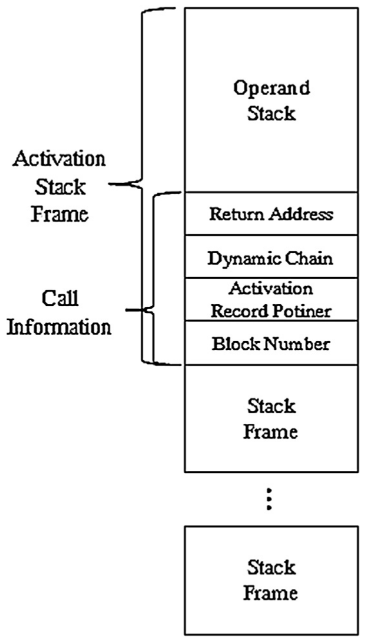

Fig. 5. Conventional stack frame structure.

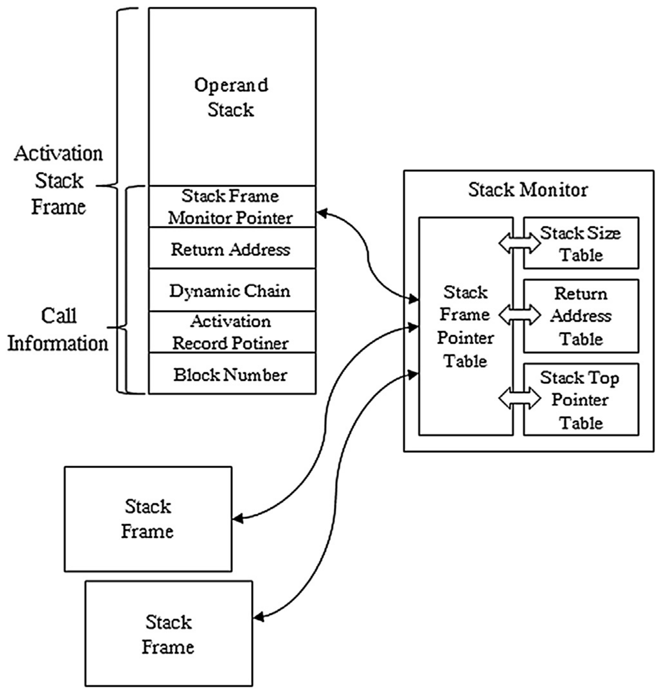

Fig. 6. Proposed stack frame structure using stack monitor.

3.3. *Virtual machine with stack monitoring*

A stack overflow is the most typical vulnerability used by hackers to attack programs and computer systems. Thus, the prevention of stack overflow attacks serves to considerably enhance the reliability of a system [24,25]. We aimed to implement secure executing environments for IoT services by proposing a VM-based stack protection technique using separated stack frames.

A general representation of the runtime stack configuration of programs is shown in Fig. 5. The memory area is shared by multiple stack frames and a new stack frame is created at every function calling. Each stack frame consists of an operand stack and the function call information.

These stack structures can easily express the relationship of the invocation of the functions; furthermore, they have advantages such as the passing of return values and efficient memory usage. On the other hand, they have disadvantages such as the exposure of sensitive information and unintended control jumps, should the stack information be compromised by a hacker [26,27].

Previous research has led to the development of techniques for solving this problem on the system level, such as return address encryption and stack guard [26,28,29]. However, these research results are not applicable to VM environments because of extensive executing performance degradation.

In this work, we use the stack monitor on the VM to protect the stack frames and reduce the loss of execution performance. A diagram of the proposed model including a stack monitor is shown in Fig. 6.

Firstly, all stack frames are isolated by other frames. Thus, compromised information at one stack frame does not affect other stack frames. Next, the return address is managed by the stack monitor, which determines whether a change of return address has occurred. Also, at this time, the stack monitor records the stack size on the caller side to verify whether the caller's stack frame has been tampered with.

Each stack frame was isolated from other stack frames as in the sandbox model used for objects/applications in management techniques. Moreover, the main attack target as return address will be duplicated on the stack monitor; thus, we can easily detect whether the address value has been subject to tampering.

Details of the stack frame management techniques using a stack monitor are presented in Table 2.

This technique does not affect the other stacks, even if the stack frame is tainted by the hacker's attacks. Also, it is impossible to jump to the point the hacker intended attacking when a change of return address has occurred. Therefore, when the attack involves an application in a virtual machine environment, the proposed

Table 2
Proposed stack frame structure using stack monitor.

1. Record the frame address to stack monitor when the stack frame is created, and write the table pointer information of the stack monitor on the stack frame. This stack frame address is cross referenced.
2. Create a new stack frame when the function is invoked, and the stack monitor writes the stack frame size and stack top pointer of the caller.
3. Record the return address of the callee on the return address table of the stack monitor.
4. At the return time of the called function, the stack monitor judges whether the frame address is damaged by comparing the monitor pointer in the activation stack frame and the frame pointer address of the stack monitor.
5. If the result is normal, the stack monitor compares the return address in the stack frame and that of the monitor.
6. If the return address matches, the stack monitor inspects the stack size of the caller by using the recorded information in the stack size table.
7. If the size is the same, the return value is copied by the stack monitor using the stack top pointer table, and the activation stack frame is changed to the caller's stack frame and the callee's stack frame is removed.
8. If problems occurred at steps 4, 5, 6, and 7 then the stack monitor regarded as the stack has been compromised and exception handling is executed.

Table 3
Weakness check result for 47 items.

ID	Fortify	Compass/ ROSE	Proposed compiler	ID	Fortify	Compass/ ROSE	Proposed compiler	ID	Fortify	Compass/ ROSE	Proposed compiler
1	○		○	17				33			○
2	○		○	18				34	○		○
3	○		○	19			○	35	○		○
4	○		○	20	○		○	36	○		○
5	○		○	21	○		○	37	○	○	○
6	○		○	22	○	○	○	38			○
7	○		○	23	○		○	39			
8	○	○	○	24	○		○	40	○	○	○
9	○			25	○		○	41			
10				26	○		○	42	○	○	○
11	○		○	27	○		○	43	○	○	○
12	○		○	28	○		○	44	○		○
13	○		○	29	○		○	45	○		○
14	○	○	○	30	○		○	46	○		○
15	○	○	○	31				47	○	○	○
16				32	○			Total	37	9	38

method can block sensitive data or system control from being obtained, and can handle exceptions for applications where the problem occurs.

4. Experimental results

4.1. Smart Cross Platform

In this section, we use the compiler we developed to experiment with the diagnosis of weaknesses that may occur in mobile applications. We selected Fortify and Compass/ROSE to compare the performance of the implemented compiler. These two selected open source software testing tools can inspect the programs written in C/C++. The secure coding rules for IoT applications used in the implemented compiler were defined during previous research and in Section 3.2.

Firstly, we determine the range of the weakness check for the implemented compiler and selected tools. Table 3 lists the checkable weaknesses from the 47 weaknesses released by the KISA (Korea Internet & Security Agency) and Ministry of the Interior [30] using three tools: Fortify, Compass/ROSE, and the implemented secure compiler. Fortify and Compass/ROSE are general software analysis tools; therefore, we selected 47 rules to ensure a reasonable comparison. The 47 weaknesses were selected because they can be applied to IoT environments.

Fortify has 37 items and Compass/ROSE has 9 items, as indicated in Table 3, and the proposed compiler checks 38 items of the total 47 items. The implemented compiler is able to cover all the different kinds of weaknesses checked by the two tools except for ID 9 and ID 32.

In addition, obtaining a false positive with the tool that checks the weaknesses is also a very important performance metric. For the 38 weaknesses that were checked by tools used in the experiment, we used the test source programs that were selected from SAMATE (Software Assurance Metrics And Tool Evaluation) [31], and each false positive result is shown in Fig. 7. The experimental results show that the proposed compiler obtained fewer false positive outcomes compared to the other tools, except for ID 36. The checkable weakness for each of the tools is Fortify 59.4%, Compass/ROSE 66.1%, proposed compiler 53.6% of the average false positives, respectively. The experimental result of the analysis showed the compiler proposed in this paper to be superior compared to the tools with which its performance is compared.

Table 4
Performance of proposed protection technique.

Test category	memcpy	memmove	socket	fget	memalloc
Original SVM	100.0%	100.0%	100.0%	100.0%	100.0%
Proposed VM with stack monitoring	104.0%	106.3%	107.5%	105.5%	106.5%

Next, to verify the efficiency of the stack monitoring, we used the SAMATE test suits for stack-based overflow and modified them to run on the proposed VM. The experimental results enabled us to confirm the ability of the exception handler to perform satisfactorily during attacks. Furthermore, the proposed VM with stack monitoring has 4%–7% overhead compared with the original SVM [17,18]. Test source program categories and execution performance rates are listed in Table 4.

5. Conclusions and further research

This paper describes tools to develop and execute secure software for IoT services. Today, the security schemes of most software rely extensively on complementary tools such as firewalls and user authentication. However, the percentage of such tools involved in software security violations accounted for only 25% of security breaches. The other 75% of security violations occurred

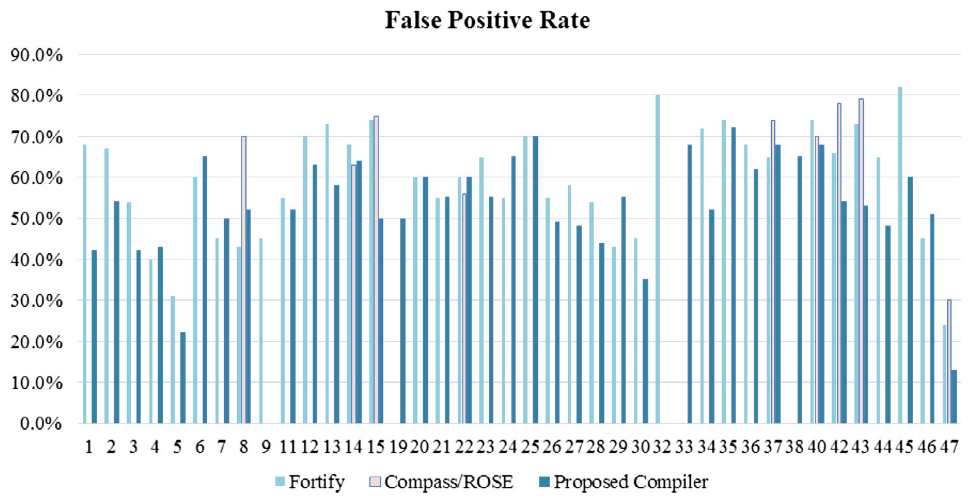

Fig. 7. False positive rate of tested analyzer (0% item means the case cannot be examined).

due to software code containing weaknesses; thus, the most effective way of enhancing the level of security is to have programmers write robust code from the start. Weaknesses in source codes can be diagnosed by the weakness analysis tools that are currently available. However, it is difficult to effectively eliminate weaknesses with this method because it requires the repeated execution of weakness analysis through a separate analyzer after correcting pre-detected weaknesses. The detection of bugs for IoT services relies mostly on classic software test methodology and classic test automation tools. This methodology separates the development process from the test process, serving as a factor that complicates problem analysis and correcting errors in the beginning of the development process.

An expanded compiler for weakness analysis and a virtual machine with stack monitoring were proposed in this study to examine the weaknesses that can exist within programs at the beginning of IoT application development and to monitor abnormal behavior of service execution. In addition, we expect the proposed compiler to expand the coverage of previous IoT service developmental platforms and reduce the cost of developing secure services.

Next, the proposed VM-based stack protection technique isolates the stack frames and monitors them using the stack monitor. In this way it is possible to block a hacker's attack aimed at overwriting the stack contents by unsanitized input values at the function call to obtain sensitive data or control of the system. This approach does not only enable the development of applications that are robust against external attacks, but it also reduces the huge cost associated with preventing problems anticipated at the service operational stage.

In future, research on automating the addition of analysis modules to compilers will be carried out. This requires the rules for secure coding to be standardized and research on automatic reading and rules analysis written in the Meta language will be carried out. In addition, there is a need to review the execution speed and the precision of the analysis results for the proposed expanded compiler. And, as a runtime of the IoT services, research on compaction of the VM, for example by minimizing the instruction set, optimizing the interpreter, and a computation offloading method using cloud services, are needed.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2013R1A2A2A01067205) and supported by a Korea University Grant.

References

[1] G. McGraw, Software Security: Building Security In, Addison-Wesley, 2006.
[2] Theresa Lanowitz, Now is the time for security at the application level, Gartner, 2005.
[3] Viega, G. MaGraw, Software Security, How to Avoid Security Problems the Right Way, Addison-Wesley, 2006.
[4] B. Chess, J. West, Secure Programming with Static Analysis, Addison-Wesley, 2007.
[5] Common Weakness Enumeration (CWE): A community-Developed Dictionary of Software Weakness Types. http://cwe.mitre.org/.
[6] SEI CERT Coding Standards: https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards.
[7] Cigital Cigital Java Security Rulepack: http://www.cigital.com/securitypack/view/index.html.
[8] K. Tsipenyuk, B. Chess, G. McGraw, Seven pernicious kingdoms: a taxonomy of software security errors, IEEE Secur. Privacy (2005) 81–84.
[9] MISRA C: http://www.misra.org.uk/misra-c/Activities/MISRAC/tabid/160/Default.aspx.
[10] H. Chen, D. Wagner, MOPS: an infrastructure for examining security properties of software, in: Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002, pp. 235–244.
[11] Plum Hall Inc. Overview of Safe-Secure Project: Safe-Secure C/C++, 2006. http://www.plumhall.com/SSCC_MP_071b.pdf.
[12] Coverity SAVE: http://www.coverity.com/products/coverity-save/.
[13] Fortify Static Code Analyzer: http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/index.html.
[14] Compass http://rosecompiler.org/?page_id=16.
[15] ROSE compiler infrastructure: http://www.rosecompiler.org/ROSE_HTML_Reference/index.html.
[16] Sparrow http://en.fasoo.com/SPARROW.
[17] Y.S. Lee, Y.S. Son, A study on the smart virtual machine for smart devices, Inf. Int. Interdiscip. J. 16 (2) (2013) 1465–1472.
[18] Y.S. Lee, Y.S. Son, A study on the smart virtual machine for executing virtual machine codes on smart platforms, Int. J. Smart Home 6 (4) (2012) 93–105.
[19] Y. Son, Y.S. Lee, Design and implementation of an objective-C compiler for the virtual machine on smart phone, Commun. Comput. Inf. 262 (2011) 52–59.
[20] Y.S Lee, Y. Son, A study on verification and analysis of symbol tables for development of the C++ compiler, Int. J. Multimed. Ubiquitous Eng. 7 (4) (2012) 175–186.
[21] Y.S. Son, S.M. Oh, Design and implementation of a compiler with secure coding rules for secure mobile applications, Int. J. Secur. Appl. 6 (4) (2012) 201–206.
[22] Open Web Application Security Project: https://www.owasp.org/index.php/Main_Page.
[23] Y. Son, I. Mun, S. Ko, S. Oh, A study on the weakness categorization for mobile applications, Korea Comput. Congr. 39 (1(A)) (2012) 434–436.
[24] R. Kumar, E. Kohler, M. Srivastava, Harbor: software-based memory protection for sensor nodes, in: ACM Proceedings of the 6th International Conference on Information Processing in Sensor Networks, 2007, pp. 340–349.
[25] A. Averbuch, M. Kiperberg, N.J. Zaidenberg, An efficient vm-based software protection, in: IEEE 5th International Conference on Network and System Security, 2011, pp. 121–128.
[26] C. Cowan, P. Wagle, C. Pu, S. Beattie, J. Walpole, Buffer overflows: Attacks and defenses for the vulnerability of the decade, in: DARPA Information Survivability Conference and Exposition, Vol. 2, 2000, pp. 119–129.
[27] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, T. Walter, Breaking the memory secrecy assumption, in: Proceedings of the 2nd European Workshop on System Security, 2009, pp. 1–8.

[28] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, H. Hinton, StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, Usenix Secur. 98 (1998) 63–78.
[29] P. Wagle, C. Cowan, Stackguard: Simple stack smash protection for GCC, in: Proceedings of the GCC Developers Summit, 2003, pp. 243–255.
[30] Secure Software Development Guides. http://www.mogaha.go.kr/frt/bbs/type001/commonSelectBoardArticle.do?bbsId=BBSMSTR_000000000012.
[31] Juliet Test Suite for C/C++. http://samate.nist.gov/SRD/testsuite.php.

YangSun Lee received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996 to 2000, a Director of Korea Multimedia Society from 2004 to 2005, a General Director of Korea Multimedia Society from 2005 to 2006 and a Vice President of Korea Multimedia Society in 2009. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system, programming languages, and embedded systems.

Junho Jeong received the B.S. and M.S. degrees in computer engineering from Dongguk University, Seoul, Korea in 2007 and 2009 respectively. He is a doctoral candidate in computer engineering at Dongguk University. His research interests include in information security system, distributed processing system, distributed and parallel algorithms, and cloud security.

Yunsik Son received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 2004, and M.S. and Ph.D. degrees from the Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 2006 and 2009, respectively. Currently, he is a Researcher of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. Also, he is a research professor of Dept. of Brain and Cognitive Engineering, Korea University, Seoul, Korea. His research areas include secure software, programming languages, compiler construction, mobile/embedded systems, and u-Healthcare.