Design and Implementation of the Compiler with Secure Coding Rules for Developing Secure Mobile Applications in Memory Usages

YunSik Son[1], YangSun Lee[2] and SeMan Oh[1]*

[1]*Dept. of Computer Engineering, Dongguk University, 26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, KOREA*

[2]*Dept. of Computer Engineering, Seokyeong University, 16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA*

sonbug@dongguk.edu, yslee@skuniv.ac.kr,
**Corresponding Author: smoh@dongguk.edu*

Abstract

With the recent dynamic growth of the mobile market, the problem of personal information leakage through mobile applications' weaknesses has become a newly rising problem. Guaranteeing the reliability of input and output data is particularly difficult nowadays because software exchange data across the internet. There is also a risk of being the target of an arbitrary intruder's malicious attack. Such weaknesses have been the root to software security violations that can cause some serious financial damages. Such weaknesses are the direct causes of software security incidents, which generate critical economic losses. Therefore it is important eliminate weaknesses in the software development stage and these areas such as the secure software development process model are being studied, recently.

In this study, a compiler which can examine applications' weaknesses at the software development stage has been designed and implemented based on existing weakness research. The proposed compiler analyzes the weaknesses within a program at the point of compilation, different to the existing development environments which separate compilers and weakness analysis tools. As a result, the new compiler enables mobile applications that are developed in rapid development cycles to be created safely from the very first stages of development.

Keywords: *Secure Coding, Secure Software, Compiler, Rule Checker, Software Verification*

1. Introduction

Recently, computer security incidents have become social hot issues and led to enormous economic losses. It has also triggered damages due to individuals' personal information being leaked. A majority of such security incidents have been directly linked to software weaknesses. Especially, the programs of today exchange data in the internet environment making it difficult to secure data reliability for the input and output data [1, 2].

For this reason a new trend proposing coding guides to solve software weaknesses at the coding stage has risen. Consequently, if weaknesses are blocked from the soft-ware development stage, the significant costs invested in recognizing and adjusting the software at the execution stage can be saved. In addition, this can contribute greatly to developing software which is safe from hackers.

For this reason a new trend proposing coding guides to solve software weaknesses at the coding stage has risen. Consequently, if weaknesses are blocked from the software

development stage, the significant costs invested in recognizing and adjusting the software at the execution stage can be saved. In addition, this can contribute greatly to developing software which is safe from hackers.

The Common Weakness Enumeration (CWE), Computer Emergency Response Team (CERT) and other organizations are carrying out research to deduct a list of weaknesses and propose an appropriate coding guide. Also, major software development companies in Korea and other countries are contributing large amounts of effort to develop higher quality software by using coding guides within each company. However, lists of weaknesses and coding guides have a general software approach, and therefore fail to consider the characteristics of the mobile platform and applications.

According to Gartner's findings, the mobile phone market is showing a rapid growth rate all over the world and this has led to smart phone related technologies to be recognized as the most magnified core technology. However, due to the expansion of the smart phone base, new serious problems such as individual information leakage through smart phone applications' weaknesses are appearing.

Currently, mobile application weakness detection uses a traditional analysis method, using the source level weakness analysis tool. By using the source level analysis tool to examine applications' weaknesses from the coding stage will enable such weaknesses to be blocked and make maintenance simpler. But the existing source level analysis tool is separated in the development stage and the testing stage, sometimes making it necessary to bring in a separate specialist to analyze the test results.

Based on the previously researched analysis methodology and tools, this study will propose a tool which combines a compiler and analysis tool so that programmers can develop safe mobile applications from the beginning stages of development.

In order to do this, firstly, we looked into the characteristics of the weakness analysis methods and analysis tools of existing secure coding. Based on this, we propose an expanded model of an ordinary compiler model which has a weakness analysis function added to it. Next, a compiler based on this expanded compiler model is created. This study introduces a compiler created based on the most frequently occurring problem, the memory usage defect problem, in real mobile devices. Lastly, the expanded compiler created is used to analyze contents and verify their validity.

2. Related Studies

2.1. Secure Coding

The software of today exchanges data in the internet environment making it difficult to secure validity of the data input and output. There exists the possibility of being maliciously attacked by random invader [1, 2]. This weakness has been the direct cause of software security incidents which generate significant economic losses.

Security systems installed to prevent security incidents from occurring, mostly consist of firewalls, user authentication system and etc. However, according Gartner's report 75% of software security incidents occur due to application programs including weaknesses. Therefore rather than making security systems for the external environment more firm, programmers creating software codes more firm is the 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.

There has been recognition of this problem recently and therefore research on se-cure coding, creating secure codes from the development stage, is being carried out actively. In CWE, a variety of weaknesses that can occur in the source code creation stage has been

analyzed and specified by the language they were written in. In addition, CERT defines the secure coding rules for creating secure source codes. Indus-tries where fatal mistakes can occur due to software defects, such as the airplane and car industry, coding rules such as Joint Strike Fighter (JSF) and Motor Industry Soft-ware Reliability Association (MISRA) Coding Rule have been implemented to con-tribute towards high quality software development.

The programming paradigm goes through gradual programming, structural programming and finally develops into object oriented programming. Along with this, the programming philosophy has developed into heightening the validity of programming from external factors so that accurate programming can result in deducing accurate results when being given accurate input. Recently, most application programs work while being connected to the network, leading to the magnification of secure coding importance for preventing security incidents and create programs safe from hackers.

Until now, security systems installed to prevent security incidents from occurring, mostly consist of firewalls, user authentication system and etc. However, according Gartner's report, as can be seen in Figure 1, 75% of software security incidents occur due to application programs including weaknesses. Furthermore, costs for making up for weaknesses are very large, so a program's security must be considered from the development stage. Consequently, rather than making the security system for external environments stronger, programmers putting more effort into creating secure software codes is the more fundamental and effective method of increasing security levels.

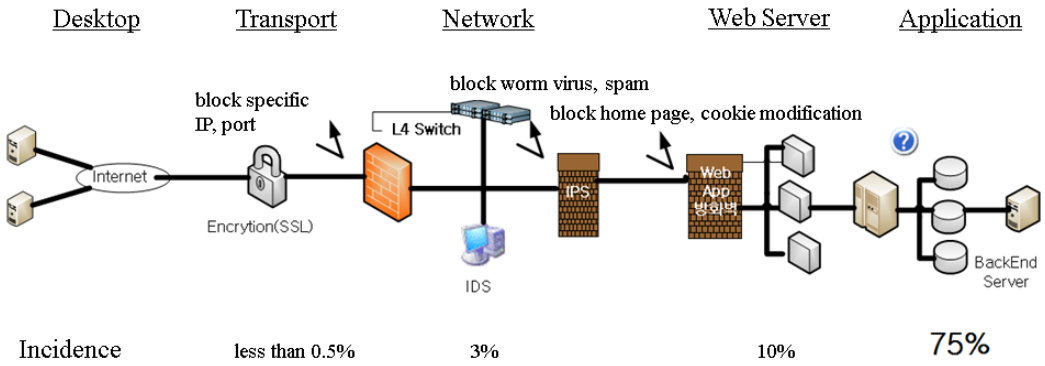

Figure 1. Incidence of Security Breaches

Recently, there has been recognition of this problem and therefore research on secure coding, writing secure codes from the development stage [3, 4], is being carried out actively. Especially, CWE (Common Weakness Enumeration) [5], an organization which 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 by the different languages. Also, CERT (Computer Emergency Response Team) defines secure coding rules to ensure secure source code creation[6]. In Cigital [7], the weaknesses can be eliminated by 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 where fatal mistakes can occur due to software defects, such as the airplane and car industry, coding rules such as JSF and MISRA Coding Rule have been implemented to contribute towards high quality software development.

Currently in countries around the world, heavy investment into lists of weaknesses, secure coding rules, analysis tools and secure systems is taking place to secure soft-ware security. Technologies related to this will become established as core technologies in future IT, software development and security industries. However, existing technologies have no consideration of the mobile environment. Especially since recent mobile applications' security problems have become serious issues, we face a situation where weakness lists, securing coding rules and weakness analysis method-ology must consider the characteristics of mobile applications.

2.2. Programming Analysis Method

The traditional program analysis method can be largely divided into static analysis and dynamic analysis. Static analysis is the method of analyzing the source code or execution file without actually running the file. Such static analysis allows analysis of source codes and its entire execution flow at a low cost, but since the program is not actually run the accuracy of analysis drops and there is a high possibility of false positives or false negatives occurring.

Dynamic analysis, unlike static analysis, is the method of analyzing a program made by software by executing each level. Because it analyzes the program by actually running it, the accuracy of analysis is high but at the same time, the analysis costs are high. In addition, the materialization complexity of the dynamic analyzer is much higher and depending on the input data, sometimes all execution paths of a program cannot be carried out. The trend of using static analysis and dynamic to find the weaknesses and bugs of recent software for commercial uses is increasing.

2.3. Source Code Weakness Analysis Tools

The source code weakness analysis tool is a tool which has been developed to automatically examine the weaknesses within a source code after it has been created by a programmer. Programmers aspire for 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 which carries out automatic analysis of weaknesses at the source code level. There exists a suitable weakness analysis method depending on each weakness and these are largely classified into static analysis and dynamic analysis methods. The static analysis method is the technology of analyzing without running the subject program and uses token, Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG) and etc. The dynamic analysis method is the technology of analyzing programs level by level while running the programs and it uses certain codes that can be used during execution time or library mapping to carry out analysis.

MOPS [9] is a model testing machine developed in 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++ [10] by Plum Hall 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 made by this software are capable of eliminating buffer overflows 100% and have less than a 5% decrease in function compared to execution files created by ordinary compilers.

Coverity's Coverity Prevent [11], is a dynamic analysis tool for source codes. Coverity Prevent 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 SCA [12] is a weakness detection tool. Fortify 360 supports C/C++, Java and ten other languages, uses static analysis and dynamic analysis to detect weaknesses of source codes. Weaknesses detected are given to the user along with statistical data.

Sparrow carries out semantic analysis to detect buffer overruns, memory leakage and other critical memory errors. It is a semantic analysis based automatic program error analyzer. Fasoo.com's Sparrow [13] provides information on the analysis time, error path and memory status on the analyzed errors.

3. Weakness for Mobile Applications

3.1. Weakness Classification

The weakness of mobile applications can be divided into the language-independent weakness, the language-dependent weakness, and the platform-dependent weakness. The language-independent weakness can appear commonly in all language because there are no linguistic characteristics: for example, naming convention, code shape, etc. The language-dependent weakness means depending on the development language of the applications. For example, applications that run on Apple's devices have the weakness related Objective-C language while Android applications have the weakness related Java language. The platform-dependent weakness depends on of runtime environment because it can appear by function supporting mobile platforms. The platform-dependent weakness can also break out by flaw of the platform. For example, Coverity Prevent found 359 bugs in Google's Android platform. 88 of the bug can lead to system crashes. Eventually, the platform bugs mean the application can be exposed to critical risk on the fly.

3.2. Method for Weakness Derivation

In the previous section, the weakness of mobile applications was divided into language-independent weakness, language-dependent weakness, and platform-dependent weakness. To derive a weakness enumeration, known weak code patterns should be collected and analyzed based on developers' experiences. However, it takes a lot of time to collect such patterns, and there is a problem of having to pass through verification processes. Therefore, it is efficient that weaknesses reflecting the features of mobile applications are derived from the already verified weakness enumeration managed in CWE and CERT. Consequently, after event-related weaknesses are derived in CWE and CERT and weakness patterns are collected based on the Android development references offered by Google, weaknesses which might happen on mobile applications are derived based on them.

3.3. Derived Weakness Enumeration

A number of derived weaknesses are 42 as Figure 2. We analyze the weakness and suggest methodology which can analyze the weakness mobile applications efficiently.

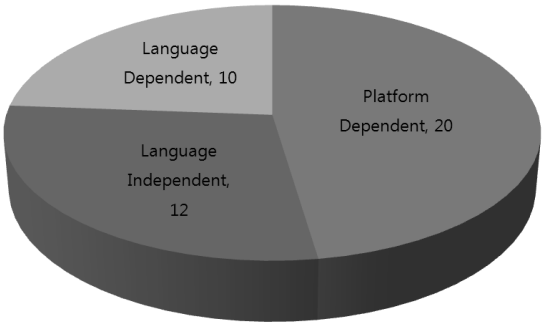

Figure 2. Derived Weakness Distribution

The derived weakness is divided into language-independent weakness, language-dependent weakness, and platform-dependent weakness by our classification methods. Figure 3 show relationship of the weakness, SCR (Standard Coding Rules), and SCG(Secure Coding Guide).

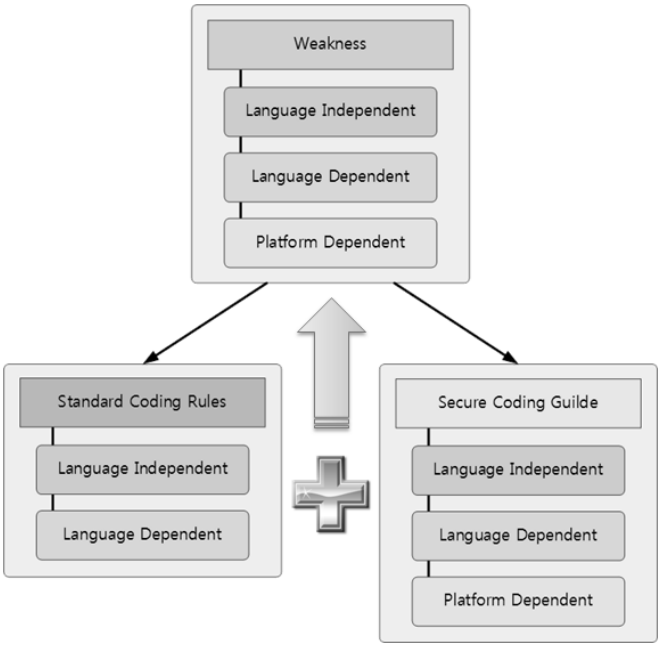

Figure 3. Relationship of the Weakness, SCR, and SCG

SCR and SCG can be suggested based-on the derived weakness enumeration. Thus, the programmer can remove the suggested weakness mostly when they develop a program satisfying SCR and SCG.

4. Design of a Proposed Compiler with Secure Coding Rules

4.1. Compiler Model

Our compiler is consisted of a module that builds traditional compilers, a static analysis module, a weakness analysis module, and a state machine. Figure 4 shows the proposed compiler.

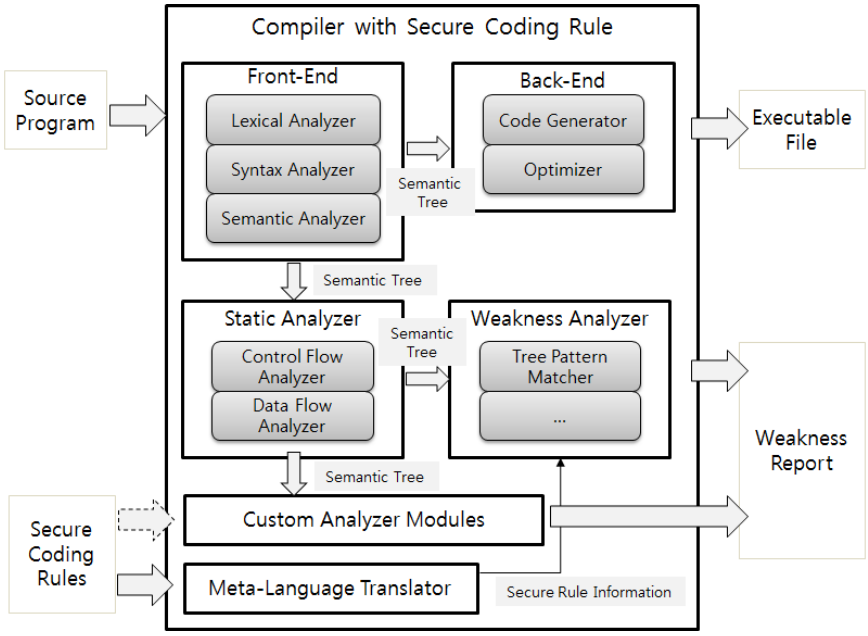

Figure 4. Proposed Compiler Configuration

In order to design the compiler with secure coding rules, a Meta language needs to be defined and a programming language's standard policy safe from the Meta language must be documented. The policy must include in addition the semantic tree [14], control flow, and data flow of the source code. The documented information is reprocessed by Meta language converter and is used as a very important piece of analysis information for weakness analysis. The static analysis module analyzes the control flow and data flow of a source program by using the symbol information and abstract syntax tree generated by the front end of the compiler. Information for analyzing weaknesses can be added to the abstract syntax tree as needed in the weakness analysis stage and this information is added to the semantic tree. Also, proposed compiler is designed to add check modules that analyze weakness point directly, if the specific weakness analysis is too difficult by meta-language method.

4.2. Secure Coding Rules in Memory Usage

Memory usage defect problems require several execution paths that occur during the execution process to be considered. Having to analyze all the possible execution paths is a task that requires high costs. This study will suggest sentence structure flow analysis as a basis memory usage defect analysis and use the results to simplify the problem at hand and propose rules for analysis.

Memory usage defects may occur due to a variety of reasons, but the main reason is lack of management of the explicitly delegated memory space by programmers. Consequently, programmers' memory use must be tracked to analyze the problem caused by memory usage.

The following two rules are for preventing memory use defects dealt with in this study.

1. Should all of the delegated memory be returned?
2. Should wrong memory reference not be used?

The problem related to the first problem is analyzing whether memory leakage is taking place. A user is explicitly delegated memory and after being used for various executions, this memory is returned and there is a possibility of memory leakage at this point. The second problem mainly deals with the problem of unexpected results occurring due to reference of wrong memory during the process of repeated delegation and deletion.

The analysis of such problems can be seen in Table 1. It includes information on entire memory space, delegated memory, and the status of allocated memory, its relation to the label in use and the degree of risk of each allocated memory.

Table 1. Notations for Memory Usage Fault Analysis

Symbol	Description
H	Total heap memory space
H_A	allocated heap memory space
S_H	state of allocated heap memory spaces
CS_H	current state of allocated heap memory spaces
S_{Li}	state of labels for the i-th allocated heap memory space
A_i	alarm level of the i-th allocated heap memory spa ce (Low/High)
S	Super vertex for all vertex, no inner edges

Table 2 includes memory use defect analysis functions. ALLOC delegates memory of specific sizes and FREE deletes delegated memory for a given address. REFL returns the memory address being referred to by a given label and on the other hand, DEREF returns all labels referring to DEREF based on the memory that is being referred to. LINKLtoL and LINKLtoA connect labels and labels, labels and memory addresses respectively. Finally, ALRAM returns the degree of risk that the current memory has.

Table 2. Memory Usage Fault Analysis Functions

Names	Description
ALLOC	size → address
FREE	address → unit
REF_L	label → address
DEREF	address → list of labels
$LINK_{LtoL}$	label, label → unit
$LINK_{LtoA}$	label, address → unit
ALRAM	address → level

Next, the results of analyzing the sentence structure flow become the basis of a usage defect analysis model for each sentence. This model consists of basic calculations, a nested block, a conditional statement, an iteration statement and function calls.

Basic calculations are composed based on sequential statements. Since only one sentence flow exists, if memory usage is tracked, entire results can be obtained. Table 3 shows the five types of calculations that can be executed in a sequential statement set. Such conditions are the minimal conditions for creating a safe program, and if they are violated memory usage defects may occur.

In the case of the first condition, reference, labels are used for reference and it is not possible for the results received to not have a memory address or not have a high degree of

risk. With the second case of dereference, there must be more than one label within all the delegated memory spaces. The third condition requires a label for one connection to not refer to several addresses or labels and for the one label it refers to, it cannot change the memory space or label reference. The fourth condition states that after memory delegation, it must be connected to a label and the last memory deletion must not be referred to by the empty memory space and the reference memory space must have a low degree of risk.

Table 3. Basic Analysis Operations

Name	Description
referencing	$REF_L(label)$ and $address \neq \Phi$ and $ALRAM(address) \neq high$
dereferencing	for all addresses of H_A, $DEREF($address$) \neq \Phi$
link	$LINK_{LtoA}$ & $LINK_{LtoL}$, relationship is N:1 for target link L, size of $DEREF(REF_L(label)) \neq 1$
memory allocation	must $LINK_{LtoA}$, after $ALLOC$
memory clearance	if $REF_L($label$) \neq \Phi$ and $ALRAM(REF_L($label$)) \neq high$ then $FREE(REF_L($label$))$

In the case of a nested block, problems related to scope and memory approach and delegation within the nested block arise. Table 4 shows the memory usage conditions for the nested block. Here it can be seen that basic calculations are used, if strayed from the block conditions are checked and if all labels defined by the block are erased and reference to the entire memory space is carried out correctly it is determined that there is no problem.

Table 4. Analysis Operation for Nested Block

Name	Description
states	1. set after state AS_H 2. set S_L^I = declared labels in inner block
block exit	1. delete S_L^I 2. for all element of AS_H, $DEREF(AS_H^i) \neq \Phi$

Two execution flows exist for conditional statements, if each flow is analyzed the results then have to be put together. Table 5 shows the conditions for analyzing conditional statements.

First the conditional statement must maintain its status set before execution, and maintain the status condition of each after execution flow. Using these three status conditions, the part affecting the entire memory status of conditional statements is deduced. Then based on basic calculations, the analysis conditions are expanded on memory delegation/deletion and label connection.

Table 5. Analysis Operation for Conditional Statements

Name	Description
states	1. set before state BS_H 2. set two after state AS_Hs
memory allocation / clearance check	1. adjustment alarm level for element of $S_H = (BS_H \; XOR \; (AS_H^1 \; U \; AS_H^2))$ 2. set current state $CS_H = BS_H \; U \; S_H$
label link check	1. set $S_L^0 = DEREF \; (BS_H)$ 2. set $S_L^i = DEREF$(each element of (AS_H^i for all element of ASH)) 3. for all j, $\{L \mid L=$element of $(S_{Lj}^0 \; U \; S_{Lj}^1 \; U \; S_{Lj}^2)\}$, if $L \in (S_{Lk}^0 \; U \; S_{Lk}^1 \; U \; S_{Lk}^2)$, $k \neq j$ then adjustment alarm level for $REF(L)$

In the case of iteration statements, similar to sentence structure analysis, repetitive flows are regarded as one set of sequential statements therefore simplifying analysis. When the 2 status sets are separated, analysis can be carried out in a similar manner to conditional statement analysis. The status state of iteration statements before execution and the status set of iteration statements bodies must be maintained to analyze the differences.

Table 6 shows the conditions and expanded analysis conditions for analyzing iteration statements.

Table 6. Analysis Operation for Iteration Statements

Symbol	Description
states	1. set before state BS_H 2. set two after state IS_H
memory allocation / clearance check	1. adjustment alarm level for element of $S_H = (BS_H \; XOR \; IS_H)$ 2. set current state $CS_H = BS_H \; U \; S_H$
label link check	1. set $S_L^0 = DEREF(BS_H)$ 2. set $S_L^I =$ DEREF each element of IS_H 3. for all j, $\{L \mid L=$ element of $(S_{Lj}^0 \; U \; S_{Lj}^I)\}$, if $L \in (S_{Lk}^0 \; U \; S_{Lk}^I)$, $k \neq j$ then adjustment alarm level for $REF(L)$

Finally, in the case of function calls, the status before the call must be recorded and the callee function status must be maintained and used for analysis. In this case of callee function analysis, the method mentioned above must be used. Callee function analysis is used to analyze cases when additional call function memory statuses are approached by callee functions, like in Table 7 and it also expands the conditions for memory delegation and deletion.

Table 7. Analysis Operation for Function Call

Symbol	Description
states	1. set before state BS_H
	2. set callee state FS_H
memory	1. $MS_H = BS_H \; XOR \; FS_H$
allocation /	2. if $MS_H = \Phi$ then skip
clearance check	3. scope resolution and
	$DEREF(element\ of\ (MS_H)) \neq \Phi$
	4. if $(BS_H \; U \; MS_H) = BS_H$ then
	scope resolution and
	$DEREF(element\ of\ (MS_H)) \neq \Phi$
	else scope resolution and
	$DEREF(element\ of\ (MS_H \cap FS_H)) \neq \Phi$
	5. $CS_H = FS_H$

Finally, when all analysis is finished, the first memory set and the memory set after shut-down can be compared to check whether memory has been leaked or not for the entire program.

5. Experimental Results and Analysis

In this chapter, a compiler based on the expanded compiler model introduced beforehand will be implemented. The new compiler has been created with the Objective C Compiler which uses secure coding rules to solve the memory usage defect problem.

In order to actually create a memory usage defect analyzer, it should take place during the semantic analysis process of the actual program or after the semantic analysis results have been collected. The semantic tree used in the Objective C compiler is in the form of a sentence structure tree and includes the semantic analysis information, making it easy to obtain analysis information.

In order to create the calculations needed for memory status analysis effectively, a bloom filter specialized for each memory status label list and membership calculation has been used to alleviate complexity.

Next is the test of example program. The source program of Fig. 5 is an example of an Objective C program related to simple memory allocation, usage and deletion. Ptr1 receives allocated memory while ptr2 refers to the allocated memory and uses it. Therefore sentence structure ① functions normally but after the first release command, sentence structure ② actually causes a memory usage defect. However, in the ordinary X-Code debugging mode, sentence structure ② uses dummy value and the run time error occurs due to the second release command call. Even the value allocated for sentence structure ③ is used to carry out the sentence, in the gnu objective C Compiler

Logically, the error must occur from sentence structure ②, however according to the compiler a dummy value may be used or it may be impossible to determine whether there is a problem or not. By using the analyzer implemented to analyze the source of Fig. 5, a warning

will appear for all sentence structure ②, the second free function call, and sentence structure ③.

```
ADT ptr1 = [[ADT alloc] init];
ADT ptr2 = ptr1;
…
// assign & using ptr1 or ptr2--- ①
…
[ptr2 release];
…
// assign & using ptr1--- ②
…
[ptr1 release];
…
// using ptr1--- ③
```

Figure 5. Example of Memory Fault

The source program of Figure 6 is an Objective-C program which has no problems logically. It is delegated and uses/deletes ptr1 and ptr2 according to the value input. However if the value is changed due to iteration statements similar to ②' s sentence structure or overflows during execution, a characteristic occurrence of C-language data, memory usage defects occur for sentence structures ③ and ④.

If it is logically right and there is no error during compilation, no problems occur during execution as well. However the characteristics of the objective C language create unpredictable insecure circumstances for programs.

By using the analyzer implemented to analyze the source of Figure 5, a warning will appear for all sentence structure ②, the second free function call, and sentence structure ③. In the case of sources of Figure 6, warnings will appear for sentence structures ①, ②, ③ and ④. This is because memory delegation for each conditional statement occurred mutually exclusively, leading to the warning level of each memory to increase. For a program which is running correctly, this may be a false warning, however considering the characteristics of the objective-C language, verification is compulsory. Furthermore, aiming towards programming with such methods is advisable.

```
ADT ptr1, ptr2;
…
// assign input
…
if (input>0)  ptr1 = [[ADT alloc] init];
else ptr2 = [[ADT alloc] init];
…
if (input>0) // assign & using ptr1--- ①
…
// iterative  code for input--- ②
// ex) while (…) input *= 100;
…
if (input>0) // assign & using ptr1--- ③
else // assign & using ptr2
…
if (input > 0) [ptr1 release];--- ④
else [ptr2 release];
…
```

Figure 6. Example of Memory Fault

6. Conclusions

Detection of bugs for smart device contents rely mostly on classic software test methodology and classic test automation tools. This methodology separates the development process and the test process, serving as a factor that makes it difficult to analyze problems and change errors in the beginning of the development process. The expanded compiler for weakness analysis proposed in this study examines the weaknesses that can exist within programs at the beginning of contents development. It also enables safe contents development and a differentiated function from existing developing/testing tools.

In the future, research on automating the addition of analysis modules to compilers will be carried out. For this, the rules for secure coding must be standardized and research on automatic reading and analyzing of rules written in Meta language will be carried out. In addition, there is a need to review the execution speed, precision of analysis results and the correlation between the two for the proposed expanded compiler.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology(No. 20110025582).

This paper was extended from the previous research paper “A Study on the Compiler with Secure Coding Rules for Smart Device's Contents” in AITS-MSA2012.

References

[1] G. McGraw, "Software Security: Building Security", Addison-Wesley, **(2006)**.
[2] J. Viega and G. MaGraw, "Software Security: How to Avoid Security Problems the Right Way", Addison-Wesley, **(2006)**.
[3] M. Howard and D. LeBlanc, "Writing Secure Code", Microsoft Press, **(2003)**.
[4] G. Hoglund and G. McGraw, "Exploiting Software: How to Break Code", Addison-Wesley, **(2004)**.
[5] Common Weakness Enumeration (CWE), "A community-Developed Dictionary of Software Weakness Types", http://cwe.mitre.org/.
[6] J. McManus and D. Mohindra, "The CERT Sun Microsystems Secure Coding Standard for Java", CERT, **(2009)**.
[7] Cigital, "Cigital Java Security Rulepack", http://www.cigital.com/securitypack/ view/index.html.
[8] K. Tsipenyuk, B. Chess and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors", Security & Privacy, IEEE, **(2005)**, pp. 81-84.
[9] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software", Proceedings of the 9th ACM Conference on Computer and Communications Security, **(2002)**, pp. 235-244.
[10] Plum Hall, Inc., "Overview of Safe-Secure Project: Safe-Secure C/C++", http://www.plumhall.com/SSCC_MP_071b.pdf, **(2006)**.
[11] Fortify Software Inc., "Fortify Source Code Analysis(SCA)", http://www. fortify.com/products/sca.
[12] Coverity, Inc., "Coverity Static Analysis", http://www.coverity.com/products/ static-analysis.html.
[13] Fasoo.com, "About Sparrow", http://www.spa-arrow.com/.
[14] Y. S. Son and Y. S. Lee, "The Semantic Analysis Using Tree Transformation on the Objective-C Compiler", Multimedia, Computer Graphics and Broadcasting, CCIS, vol. 262, **(2011)**, pp. 60-68, Springer.
[15] B. Chess and J. West, "Secure Programming: With Static Analysis", Addison-Wesley, **(2007)**.

Authors

Yunsik Son

He 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. His research areas include smart system solutions, secure software, programming languages, compiler construction, and mobile/embedded systems.

YangSun Lee

He 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 1996-2000, a Director of Korea Multimedia Society from 2004-2005, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea Information Processing Society from 2006-2010 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of Smart Developer Association from 2011-2012. Currently, he is a Professor of Dept. of

Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.

Seman Oh

He received the B.S. degree from the Seoul National University, Seoul, Korea, in 1977, and M.S. and Ph.D. degrees from theDept. of Computer Science, Korea Advanced Institute of Science and Technology, Seoul, Korea in 1979 and 1985, respectively. He was a Dean of the Dept. of Computer Science and Engineering, Graduate School, Dongguk University from 1993-1999, a Director of SIGPL in Korea Institute of Information Scientists and Engineers from 2001-2003, a Director of SIGGAME in Korea Information Processing Society from 2004-2005. Currently, he is a Professor of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.