Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.5, Oct. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.5.973

코딩 표준 검색 기법을 이용한 소프트웨어 보안 취약성 검출에 관한 연구

장 영 수†‡
한국폴리텍대학

A Study on Software Security Vulnerability Detection Using Coding Standard Searching Technique

Young-Su Jang†‡
Korea Polytechnic

요 약

정보 보안의 중요성은 응용 소프트웨어의 보안으로 인해 국가, 조직 및 개인 수준에서 점점 더 강조되고 있다. 임베디드 소프트웨어를 포함하는 높은 안전성 소프트웨어의 개발 기술은 항공 및 원자력 분야 등 에 국한되어 사용되었다. 하지만 이러한 소프트웨어 유형은 이제 응용 소프트웨어 보안을 향상시키는 데 사용된다. 특히 보안 코딩은 방어적 프로그래밍을 포괄하는 개념으로 소프트웨어 보안을 향상시킬 수 있다. 본 논문에서는 개선된 코딩 표준 검색 기법을 적용한 소프트웨어 보안 취약성 탐지 기술을 제안한다. 공개된 정적 분석 도구는 소프트웨어 보안 가능성을 분석하고 취약점을 유발하는 명령어를 분류하는 데 사용되었으며, 소프트웨어 취약점을 유발할 수 있는 API 및 버그 패턴을 쉽게 감지하여 향상시킬 수 있다.

ABSTRACT

The importance of information security has been increasingly emphasized at the national, organizational, and individual levels due to the widespread adoption of software applications. High-safety software, which includes embedded software, should run without errors, similar to software used in the airline and nuclear energy sectors. Software development techniques in the above sectors are now being used to improve software security in other fields. Secure coding, in particular, is a concept encompassing defensive programming and is capable of improving software security. In this paper, we propose a software security vulnerability detection method using an improved coding standard searching technique. Public static analysis tools were used to assess software security and to classify the commands that induce vulnerability. Software security can be enhanced by detecting Application Programming Interfaces (APIs) and patterns that can induce vulnerability.
Keywords: Information Security, Secure Coding, Defensive Programming, Public Static Analysis tools

I. Introduction

The quality of program code influences a variety of factors, including program security, repair, and maintenance. Most program errors stem from unexpected input values and the debugging of errors requires substantial personal and physical resources[1]. It is therefore essential to predict and trace the location of bugs that

Received(07. 09. 2019), Modified(08. 31. 2019), Accepted(09. 02. 2019)
† 주저자, jyskkh@naver.com
‡ 교신저자, jyskkh@naver.com(Corresponding author)

arise throughout the course of a program's operation because debugging is necessary to minimize the damage when uncontrollable or unexpected errors occur[2].

Defensive programming is a coding standard aimed at defining a coding style and format applied during the development of a program. Security vulnerabilities identified while developing a program or designing a compiler should be resolved to make the code more concise and easier to read, as well as to reduce the possible occurrence of bugs. Such a defensive programming technique gives first priority to the protection of a program from inappropriate input values, so that errors are exposed as they occur in the course of program execution[3]. Defensive programming facilitates the modification of code that produces errors. Furthermore, it prevents other programs and the entire system from being damaged when errors occur in the code.

In this study, we improve the security of applications using an improved secure coding detection technique. In order to achieve this, we divide the process of software vulnerability analysis into a vulnerability monitoring stage and a vulnerability processing stage. The first stage is focused on extracting vulnerability-inducing functions while the second stage is focused on processing extracted function vulnerability. The process is followed by the improvement of vulnerability-inducing functions and vulnerability categories.

The remainder of this paper is organized as follows: Section Ⅱ reviews previously reported studies. Section Ⅲ details our security vulnerability processing model, Section Ⅳ presents the evaluation process, and Section Ⅴ discusses our rum-time measures. Section Ⅵ presents a comparison, and finally, Section Ⅶ presents our conclusions.

Ⅱ. Literature review

A static analysis is commonly performed without actually executing the software. It examines applications on a non-runtime basis, which is advantageous in assessing applications and thereby detecting errors in the input and output of the software.

Flawfinder tool generally provides a database of application programming interface (API) symbols related to the vulnerability. This database can be used to evaluate vulnerability detection through static code analysis. We use a similar approach to define the values of the authorized function text.

Zampetti *et al.*[4] used a flow-sensitive and inter-procedural system for the discovery of software vulnerabilities through a bottom up analysis of procedures, basic blocks, and the whole program. Typical precision issues with static analysis, especially when dealing

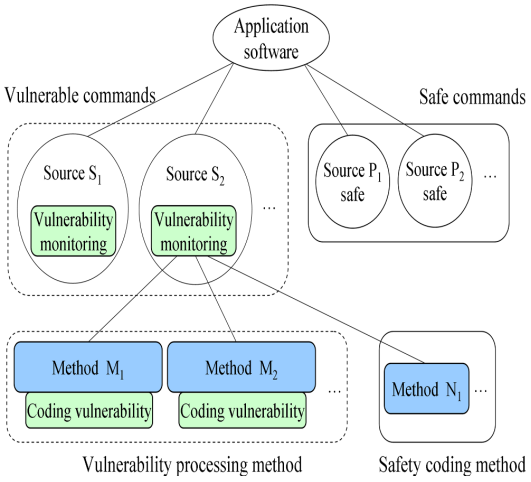

Fig. 1. Two-Level Processing Model (TLPM) for software vulnerability analysis.

with dynamically constructed strings, mean that they may identify several such illegal flows, even if these paths are infeasible.

Nagy *et al.*[5] applied the Java String Analyzer to extract models of program's database queries. They used these models as the basis for a runtime monitoring and protection for SQL injection attacks.

Jang and Choi[6] propose a technique to handle vulnerabilities by combining static and dynamic analysis techniques. The authors focus on the sanitization process and abstract away other detail of the application. Especially they created behavior information for extracted paths from the program control path.

III. System model

3.1 Analysis model

In this section, we detail our software security analysis model. The Two-Level Processing Model (TLPM) complies with a collection of guides and rules determined based on the requirements of the project and organization, as recommended by CERT/CC[1] in accordance with secure coding standards. TLPM follows the general coding standard rules for software security analysis. In particular, through the use of secure coding standards, it is possible to classify coding errors, which are considered the main causes of software vulnerability. We use two analysis models, namely security vulnerability monitoring and security vulnerability analysis processing, to analyze software vulnerability, as seen in Fig.1. In the first stage of security vulnerability monitoring, coding vulnerability-inducing functions are discerned. In this stage, the meta-data for vulnerability-inducing functions is created to be processed in the second stage. In the second stage of security vulnerability analysis processing, coding vulnerability-inducing functions are distinguished from functions that do not induce vulnerability.

3.2 Model implementation

3.2.1 Coding standard example

The following is an example of the coding standards for a C/C++ and Java program with our method applied.

(1) Declaration and initialization:
typedef is used instead of define, and *const* is attached to a constant value for protection against possible modifications.

(2) Expression:
"()" is used in consideration of the order of priority for operators, and *sizeof*() needs to be used at the time of integer size assignment.

(3) Memory:
The memory is assigned and cleared in the same module at the same block level, and it needs to be initialized as Null after the *free*() pointer validation function is used. The cleared memory is not re-accessed.

(4) Character and string:
strncpy() is used instead of *strcpy*(); *strncat*() instead of *strcat*(); *fgets*() instead of *gets*(); and snprintf() instead of *sprintf*(). The data to be copied is checked to determine whether it is Null or exceeds the buffer size.

1) https://www.sei.cmu.edu/

```
Function Types {
    None = 0,
    malloc = 1,
    strchr = 2,
    ...
    gets = 54,
    strcpy = 55,
    sprintf= 56
}
```

Fig. 2. Example of security- vulnerable functions for C.

Example Rule 1: Conditional attempts.
(?:\s+[\d\w]\s?=)|(?:if\s?\([\d\w]\s?=)

Example Rule 2: Basic authentication bypass attempts.
(?:&&\s*(?:||)?\s*[([]\s*!)|(?:&\s*"\%)|(?:|\s*"\s*\W*["\d])|(?:"\s*(?:n?and|x?or|not|\|\||\&\&)\s+[\s\w]+=\s*\w+\s*--)|(?:"\s**\s*\w+\W+")|(?:"\s*[^?\w\s=.,;\/)(]+\s*[(@]*\s*\w+\W+\w)

Example Rule 3: Query attempts.
?:(select|:)\s+(?:benchmark|if|sleep)

Fig. 3. Example of internal database rulesets for C and Java.

3.2.2 Vulnerable function extraction

We utilize a suffix array-based technique[7] to extract vulnerable functions as shown in Fig.2. Fig.3 shows an example of the internal database rulesets used for detecting vulnerabilities. Specifically, we use a phrase discovery algorithm[8] that employs a variant of suffix arrays extended with an auxiliary data structure. We remove single words that are comments and phrases beginning or ending with stop words. We inquire into words that are single words and phrases. Additionally, removing safe functions can reduce the calculation complexity. Brief operations of the TLPM and the working scheme for our model are presented in Fig.4. We define our model as follows:

$$\mathrm{TLPM}(M,\ S,\ KW,\ \lambda)$$

where M is a set of secure coding standard,

- S is a set of source programs,
- KW refers to the extraction of vulnerability-inducing keywords,
- $\lambda(\lambda \in [0,1])$ is used to obtain a tradeoff score between vulnerable and safe functions.

Set KW extracts the keywords for vulnerability-inducing functions, and λ distinguishes vulnerable and safe functions.

TLPM provides a set SP of vulnerability function-ranked lists to perform vulnerability analysis as follows:

$$\lambda R(s_i, m) + (1-\lambda) PNs(s_i) \tag{1}$$

where $R(s_i,\ m)$ is the relevance score for source program s_i to secure the coding standard m, and $PNs(s_i)$ is the facet novelty score of source program s_i.

At the beginning of TLPM, no source program is selected (i.e., $SP = \emptyset$), and no vulnerability-inducing function is selected (i.e., $KW_c = \emptyset$) either. TLPM selects the

```
Procedure TLPM(M, S, KW, λ):
 1. SP = Ø
 2. KW_c = Ø
 3. select the first source program s* (s* ∈ S)
 4. while TRUE do
 5.     SP = SP ∪ {s*}
 6.     S = S \ {s*}
 7.     KW_c = KW_c ∪ KW_s*
 8.     if || KW_c|| == || KW || then
 9.         s* = arg max(λR(s_i,m)+(1-λ)PNs(s_i))
10.         break
11.     end
12.     select the next source program s* (s* ∈ S)
13. end
14. return SP
End Procedure
```

Fig. 4. A brief scheme for TLPM.

first source program s^*, and then the set KW_{s^*} of keywords selected by s^* is added to the vulnerability-inducing function set KW_c. Afterward, the facet novelty score representing the vulnerability value of the vulnerability-inducing function is calculated, and the source program with the largest vulnerability score in Equation (2) is selected at the second position. Specifically, since the source programs cover many different types of vulnerability-inducing functions, we argue that there exists a set of keywords whose subsets can accurately represent different underlying facets. We can therefore select a source program subset to cover the keywords, so as to cover all the facets. This procedure is repeated until all vulnerable functions are identified. The vulnerability score indicates the riskiness of the vulnerable function and range from 0~1, with 0 indicating no vulnerability and 1 indicating high vulnerability.

In order to identify the facets of security-vulnerable functions, we define security-vulnerability facets based on the keyword coverage. Table 1 shows the summary of notations for the novelty of each facet. Suppose a set KW of q keywords is extracted from n vulnerability-inducing functions, different underlying facets will contain different keyword subsets:

$$FS(kw_j, kw_w) = \sum_{i=1}^{n} (FK(kw_j, s_i) - FK(kw_w, s_i)) \quad (2)$$

where $i \in [1,n]$, $j,w \in [1,q]$, and $FK(kw_j,s_i)$ represents the frequency of vulnerability-inducing functions for the keyword kw_j appearing in the source program s_i:

Table 1. Summary of notations for facets.

Notations	Description
$FS(kw_j,kw_w)$	The security-vulnerable facet of keywords (kw).
$FK(kw_j,s_i)$	The frequency of a keyword (kw) in the source code(s).
$t(kw_j,s_i)$	The number of times a keyword (kw) appears in the source code(s).
$PN_w(kw_j)$	The representative power of a keyword (kw) for a source code(s).
$FS'(kw_j,KW_c)$	The keywords (kw) covered by a vulnerable command from a set KWc.
$V\text{-}cmd$	The vulnerability analysis performance.

$$FK(kw_j,s_i) = t(kw_j,s_i) / \sum_{w=1}^{q} t(kw_w,s_i) \quad (3)$$

where $t(kw_j,s_i)$ represents the number of times a keyword kw_j appears in source program s_i. A larger facet score between two keywords indicates a higher degree of novelty.

In the secure coding standard, the novelty of a command keyword is defined as the degree of dissimilarity between a command keyword and a vulnerability-inducing command keyword set KW_c as below:

$$PN_w(kw_j) = FS'(kw_j, KW_c) \quad (4)$$

where $w,j \in [1,q]$ and $FS'(kw_j,KW_c)$ represent the keywords covered by the vulnerable command from a set KW_c of q keywords. The novelty between keyword kw_j and KW_c in a source program is defined as the distance between kw_j and any keyword in KW_c:

$$FS'(kw_j,KW_c) = FS'(kw_j,KW_c \setminus kw_j) \quad (5)$$

We select set S from the source program with the largest vulnerability score of

relevance and importance for the secure coding standard. We extract the current state of vulnerability-inducing commands based on the riskiness score from m.

$$s^* = \arg\max_{s_i}(\lambda R(s_i, m) + (1-\lambda) PN_s(s_i)), \forall s_i \in S \quad (6)$$

where $R(s_i, m)$ is the relevance score of source program s_i for the secure coding standard. We use the frequency to measure the representative power of a command keyword for a source program, since it is accepted that keywords with frequent occurrence in a source program have a strong relational power for that source program. Finally, the facet novelty score of source program s_i is given by:

$$PN_s(s_i) = \sum_{j=1}^{q} FK(kw_j, s_i) \times PN_w(kw_j) \quad (7)$$

3.2.3 Security vulnerability analysis

To evaluate the vulnerability analysis performance, we used Command Keyword Retrieval[9]. Let $stamethod(s_i)$ be the set of command keywords for which s_i is relevant. n is the total number of command keywords. Command Keyword Retrieval at rank r is defined as the percentage of command keywords covered by source programs:

$$V\text{-}cmd = \frac{|\bigcup_{i=1}^{r} stamethod(s_i)|}{n} \quad (8)$$

A larger *V-cmd* value indicates greater keyword coverage. When all keywords are covered, the value of *V-cmd* will equal 1. For any vulnerability level, we can find a minimal optimal rank so that the Command Keyword Retrieval value at that rank is equal to 1. When different command keywords have different threat levels, such as the vulnerability level, *V-cmd* measures can show which command keyword is the most vulnerable command keyword[10].

IV. Evaluation

4.1 Evaluation measures

We used public static analysis tools to measure vulnerabilities instead of relying on reported vulnerabilities. Unlike the process of manual code review or penetration testing, which produces reported vulnerabilities, static analysis is a repeatable and scalable technique for measuring vulnerabilities[11].

Flawfinder[2] analyzes C/C++ source code to identify vulnerabilities using an internal database for the ruleset.

To determine the accuracy of the experiment, we used the Rough Auditing Tool for Security (RATS[3]) program, a static analysis tool that identifies vulnerabilities in C/C++ source code.

FindBugs[4] is another static analysis tool. FindBugs analyzes Java source code to determine the patterns of bugs (as well as program bugs) and suggests modifications while conducting vulnerability searches with regard to the compiled byte code.

We employed public static analysis tools to verify whether the software was developed without coding errors. In our system model, these tools are used to identify false positive errors and extract vulnerable functions and patterns from application programs.

2) http://www.dwheeler.com/flawfinder
3) http://www.estima.com/ratsmain.shtml
4) http://findbugs.sourceforge.net

Table 2. Results of the vulnerability evaluation using the Juliet test suite.

Vulnerability	CWE	Program language	LOC	Test-cases		TLPM Detection	Native tools Detection
				Illegit set	Legit set		
Buffer handling	CWE-190	C/C++	2,193	84	21	Success	Success
	CWE-191	C/C++	1,590	92	13	Success	Fail
	CWE-120	Java	2,291	161	29	Success	Fail
	CWE-121	Java	1,892	109	31	Success	Success
	CWE-131	C/C++	2,321	114	21	Success	Fail
Input validation	CWE-129	Java	930	20	10	Success	Fail

4.2 Empirical test suites

To evaluate the TLPM verification results, we used the Juliet test suite[12] for C/C++ and Java software. The cases cover 181 different Common Weakness Enumeration (CWE)[5] entries. We employed the Juliet test suite to verify whether TLPM can successfully detect faults. The illegit suite includes over 50 different illegal cases like buffer overflows and memory leaks. These inputs contain data that can destroy the input validation process for a program. Table 2 summarizes the results of vulnerability detection using the Juliet test suite. "Buffer handling" and "input validation" are the most common security issues in programs, hence the problems in this area are the most common test cases in C/C++ and java. The third column lists the program language. The fourth column lists the Line of Code (LOC). The Test Cases column shows the number of test cases included in each set. The "TLPM Detection" column shows the detection results obtained with the TLPM method. The final column shows the detection results obtained with the public static analysis tools (i.e., Flawfinder, RATS, FindBugs). Public static analysis tools can detect "CWE-190 (integer overflow)" and "CWE-121 (stack based buffer overflow)", but they do not detect "incorrect buffer size calculations" and "validation of array index".

4.3 Real-world test suites

We used real-world Customer Order Service (COS) programs. These programs are used in the real world. Taking into consideration program approachability, we used online applications written in C and Java. For batch programs, we chose programs written in C rather than Java because the programs need to transfer data to external systems. The numbers of programs used were:

(1) Online application C programs: 55 (concurrent program users: 30 users),
(2) Batch handling C programs: 10 (number of handling: 50,000~80,000 cases per day), and
(3) Online application Java programs: 100 (concurrent program users: 30 users).

5) https://cwe.mitre.org

Table 3. Example of fixed size local buffer vulnerable function detection count from FlawfinderTLPM and RATS_TLPM in C/C++.

Vulnerability categories	Vulnerable functions	FlawfinderTLPM vulnerable function detection count	RATS_TLPM vulnerable function detection count
Fixed size local buffer	strcat	4,312	4,012
	strcpy	3,219	3,121
	sprintf	695	598

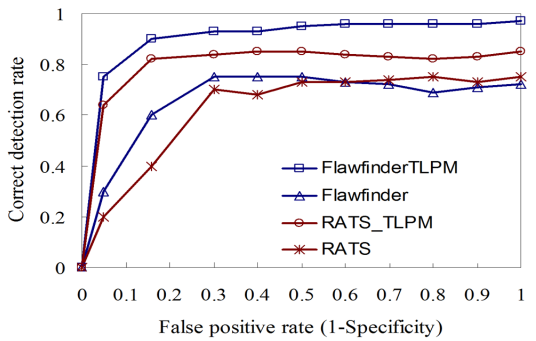

Fig. 5. Receiver operating characteristic (ROC) curves comparing the accuracy of vulnerability-inducing command detectors, one using the native detector and the other with TLPM.

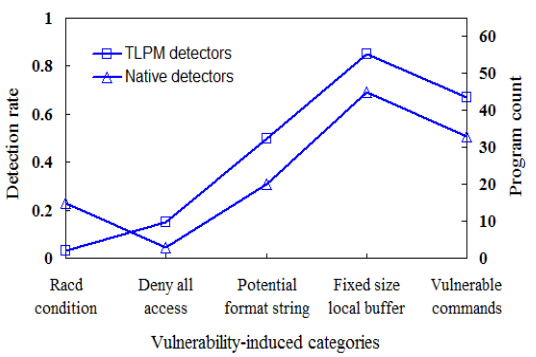

Fig. 6. Vulnerability-inducing category detection for C programs.

4.4 Evaluation result

In our model, the vulnerability of the programs was determined by evaluating vulnerability-inducing functions and the pattern of bugs. TLPM was applied to static analysis tools for vulnerability analysis.

4.4.1 Security vulnerability analysis for C programs

Table 3 shows an example of the fixed size local buffer vulnerable functions of C/C++ programs extracted from security vulnerability analysis. The strcat() function achieved the best keyword retrieval, though there was no significant performance difference among the methods. The strcpy() function had the second best results with nearly similar performance.

Fig.5 shows the results of vulnerability-inducing function discernment for the programs evaluated. The receiver operating characteristic (ROC) curves show that FlawfinderTLPM[6] and RATS_TLPM[7] reduced the number of false positives significantly. From the diagram, we see that FlawfinderTLPM and RATS_TLPM Receiver Operating Characteristic (ROC) curves have a very sharp rise from (0:0). Again, the detector using the TLPM method is more accurate over a wide range of false positive values. The vulnerability of C programs was classified into categories for race condition, deny all access, potential format string, fixed size local buffer, and vulnerable commands

6) FlawfinderTLPM indicates the application of the TLPM method.
7) RATS_TLPM indicates the application of the TLPM method.

with variable input values to conduct vulnerability-related analyses. Fig.6 shows the detection rates for vulnerability-inducing categories. Among the results, the fixed size local buffer is ranked number one. If a fixed size local buffer (e.g., an array) is used in an unsafe manner, overflowing the buffer allows an attacker to overwrite the Return Instruction Pointer (RIP) with an arbitrary value[6]. Ranked at number two is the vulnerable command with variable input values. If a variable-length string value is copied into a buffer that is too small to hold it, the string is truncated abruptly to fit the buffer. No attempt is made to trailing-align or trim the value, although strings are null-terminated.

4.4.2 Security vulnerability analysis for Java programs

Table 4 shows an example of methods that can induce vulnerabilities in Java programs. The "Doddy" category has the highest keyword retrieval. The category of "Bad practice" is second, with close performance for "Security".

The vulnerability of Java programs was distinguished based on the pattern of bugs[13] using FindBugsTLPM[8]. Similar to the analysis of C programs, Fig.7 shows the patterns of bug discernment in the Java application programs analyzed. The ROC curves show that FindBugsTLPM significantly reduced the number of false positives. FindBugsTLPM detected program bugs and determined the pattern of bugs to provide guidelines for modification and to search for vulnerabilities in the compiled byte codes. The vulnerability patterns for Java programs were classified

8) FindBugsTLPM indicates the application of the TLPM method.

Table 4. Example of vulnerable category detection count for FindBugsTLPM in Java.

Vulnerability categories	Findbugs vulnerable category detection count
Dodgy	109
Bad practice	82
Security	71
Malicious code	30

into categories for correctness (COR), bad practice (B.P), experimental (EXP), internationalization (INT), malicious code vulnerability (M.C.V), multithreaded correctness (M.C), performance (PER), security (SEC), and dodgy (DOD) to conduct vulnerability-related analyses.

Fig.8 shows the results of

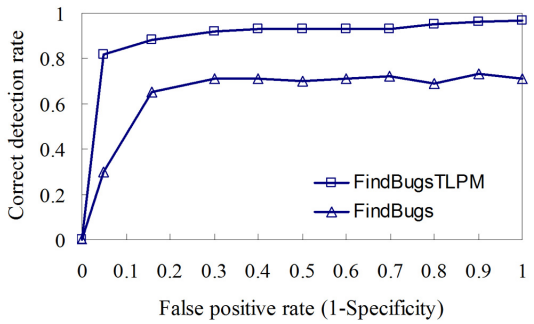

Fig. 7. ROC curves comparing the accuracy of vulnerability-inducing pattern detectors, one using the native detector and the other with the TLPM.

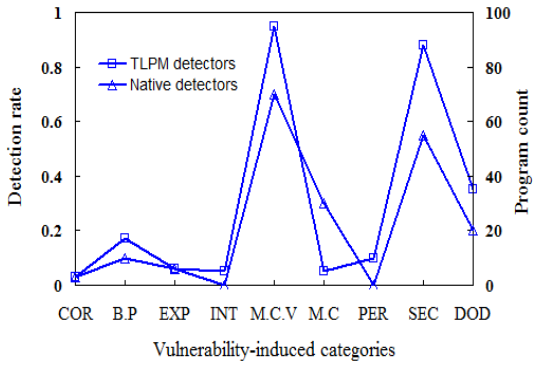

Fig. 8. Vulnerability-inducing category detection for Java programs.

vulnerability-inducing pattern detection according to the categories. For M.C. in particular, TLPM detectors had a lower detection rate than native detectors. This is because of incorrect initialization and update of the static field in the source code. When we modified the field to initialize the object, the other thread could not access the stored object until the thread was completely initialized.

V. Run-time measures

We examined the run-time of our proposed model. Fig.9 shows the average test performance of TLPM as the number of test examples is varied. We saw a substantial improvement in performance as the test set size increased. Fig.10 shows the actual run-time for the different processes executed in our implementation of TLPM in the source programs. We divided our implementation into three parts: preprocessing, vulnerability command extraction, and secure coding application. The preprocessing step had the longest run-time, 150 milliseconds on average; secure coding application had the second longest run-time, taking on average 105 milliseconds; vulnerability command extraction took 45 milliseconds. This means our implementation of TLPM takes took than 350 milliseconds to analyze vulnerabilities in the practical application source programs.

VI. Conclusion and future work

This paper proposes a novel TLPM that uses improved secure coding techniques to reduce the number of security vulnerabilities in software. This model, along with a secure coding standard, will be useful in enhancing software security. In this paper, we only provide a simple implementation of our prototype. Therefore, there is substantial room to improve the identification of application vulnerabilities with TLPM using other vulnerable-function extraction models. Furthermore, we can also improve the time effectiveness and extraction accuracy for our implementation.

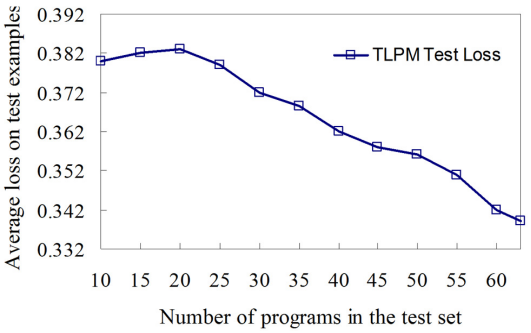

Fig. 9. The average performance of TLPM.

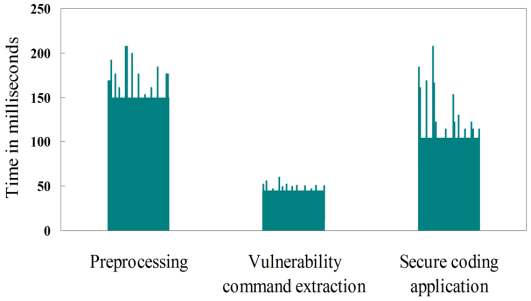

Fig. 10. Run-time for steps of TLPM implementation.

References

[1] "APCERT Annual Report 2017", Carnegie Mellon University software engineering institute, 2018.

[2] J. Stark, "Product lifecycle management," In Product Lifecycle Management, Springer, pp. 1-29, Dec. 2015.

[3] P. Nunes, I. Medeiros, J. Fonseca, N.

Neves, and M. Correia, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," Computing, vol. 101, no. 2, pp. 161-185, Sep. 2019.
[4] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Penta, "How open source projects use static code analysis tools in continuous integration pipelines," In IEEE/ACM 14th International Conference on MSR, pp. 334-344, May. 2017.
[5] G. Nagy and A. Cleve, "A static code smell detector for SQL queries embedded in Java code," IEEE, pp. 147-152, Apr. 2017.
[6] Y.S. Jang and J.Y Choi, "Automatic prevention of buffer overflow vulnerability using candidate code generation," IEICE TRANSACTIONS on Information and Systems, pp. 3005-3018, Dec. 2018.
[7] T. Gagie, G. Manzini, and D. Valenzuela, "Compressed spaced suffix arrays," Mathematics in Computer Science, pp. 151-157, May. 2017.
[8] N. Zhong, Y. Li, and S.T. Wu, ""Effective pattern discovery for text mining," IEEE transactions on knowledge and data engineering, vol. 24, no. 1, pp. 30-44, Apr. 2010.
[9] W. Webber, "Evaluating the Effectiveness of Keyword Search," IEEE Data Eng. Bull, vol. 33, no. 1, pp. 54-59, Dec. 2010.
[10] S.S. Kia, B.V Scoy, B.J. Cortes, R.A. Freeman, K.M. Lynch, and S. Martinez, "Tutorial on dynamic average consensus: the problem, its applications, and the algorithms," IEEE Control Systems Magazine, vol. 39, no. 3, pp. 40-72, Apr. 2019.
[11] J. Bleier, E. Poll, H. Xu, and J. Visser, "Improving the usefulness of alerts generated by automated static analysis tools," Oct. 2017.
[12] NIST, "Juliet test suite" https://samate.nist.gov/SARD/testsuite.php, Jun. 2019.
[13] G. Balan and A.S. Popescu, "Detecting Java Compiled Malware using Machine Learning Techniques," IEEE, pp. 435-439, May. 2018.

〈저자소개〉

장 영 수 (Young-Su Jang) 정회원
2011년 2월: 고려대학교 소프트웨어공학과 석사
2019년 8월: 고려대학교 컴퓨터·전파통신공학과 박사
2017년 12월~현재: 한국폴리텍대학 스마트소프트웨어학과 조교수
〈관심분야〉 소프트웨어 보안, 프로그래밍 언어, 정형기법