Journal of KIIT. Vol. 17, No. 9, pp. 133-144, Sep. 30, 2019. pISSN 1598-8619, eISSN 2093-7571 133
http://dx.doi.org/10.14801/jkiit.2019.17.9.133

Source Code Instrumentation Technique for Buffer Overflow Vulnerability Detection

Young-Su Jang*

Abstract

The requirements of application software are becoming increasingly complex, and the importance of information security is increasingly emphasized at the national, organizational and individual levels. In this paper, we designed and implemented the Semantic Function Libraries (SFL) white box vulnerability detection library. The SFL dynamically locates sensitive Application Programming Interface (API) function calls in C/C++ to detect code vulnerabilities at program execution and check source code variable safety conditions. Our research approach is that instrumentation checks are performed before working with sensitive API functions to infer various buffer sizes used in C/C++ source code and to store information. In addition, the analysis of the check is based on semantic dynamic analysis based on API functions. Our proposed technique is useful for previewing software monitoring and for predicting and tracking the location of bugs that occur during program operation.

요 약

어플리케이션 소프트웨어의 요구 사항은 갈수로 복잡해지고 있으며, 정보 보안의 중요성 또한 국가, 조직 및 개인 수준에서 점차 강조되고 있다. 본 논문에서는 Semantic Function Libraries (SFL) 화이트 박스 취약성 발견 라이브러리를 설계 하고 구현하였다. SFL은 C/C++에서 민감한 Application Programming Interface (API) 함수 호출을 동적으로 찾아 프로그램 런타임 시에 코드 취약성을 탐지하고 소스코드 변수 안전성 조건 검사를 수행한다. 우리의 연구 접근 방법은 C/C++ 소스코드에서 사용되는 서로 다른 버퍼 크기를 추측하고 정보를 저장하여, 민감한 API 함수 작업을 수행하기 전에 인스트루먼트 검사를 먼저 수행한다. 또한 검사에 대한 분석은 API 함수 기반의 의미론적 동적 분석에 기반을 두었다. 우리가 제안하는 기술은 미리보기 소프트웨어 모니터링과 프로그램 작동 과정에서 발생하는 버그의 위치를 예측하고 추적하는 데 유용하다.

Keywords
buffer overflow, information security, sensitive operation, look-ahead software monitoring, software quality evaluation

Ⅰ. Introduction

C/C++ is a widely used programming language for operating systems, critical software, and system software (e.g., e-mail and domain name system servers on the Internet). C/C++ developers often use memory allocated arrays, which are allocated and deallocated automatically without developer intervention. Because

* Dep. of smart software, Korea Polytechnic
- ORCID: https://orcid.org/0000-0002-1963-1852

· Received: Aug. 23, 2019, Revised: Sep. 00, 2019, Accepted: Sep. 00, 2019
· Corresponding Author: Young-Su Jang
Department of SmartSoftware, Korea Polytechnics, 41-12, Songwon-gil, Gongdo-eup, Anseong-si, Gyeonggi-do, 17550, Korea
Tel.: +82-31-650-7257, Email: jyskkh@naver.com

of their convenience, memory allocated arrays are used to store data from external inputs in the Application Programming Interface (API) function (hereinafter called “function”) calls. However, memory corruption vulnerabilities (e.g., stack allocated overflow) are found most commonly in C/C++ applications[1]. Although memory corruption vulnerabilities have received substantial attention, most solutions have failed to address the problem fully[2]. Furthermore, attackers are constantly searching for new exploits[3][4]. Most solutions presume that if secure coding rules or built-in verification functions are used for a program code, then the result of that program is safe[4]-[6]. However, if certain malicious values are supplied to the sanitization function, this does not completely protect against all attacks[3][7]. In general, it is well-known that both static analysis and dynamic analysis must be used to verify software weaknesses and for test generation. Each technique provides different solutions in terms of reliability, speed, and precision. Static analysis provides all potential execution paths but requires some heuristic for selecting only the relevant paths[8]-[10]. Therefore, it can cause false negatives. Dynamic analysis incurs high computing overhead and cannot guarantee that all possible execution paths are exercised[11][12]. Therefore, it has limited defect detection coverage.

In this paper, we propose a look-ahead dynamic technique that gives first priority to program protection from inappropriate input values so that errors are exposed as they occur in the course of executing a C/C++ application. This is based on the semantic checking of whether a statement follows according to the developers’ intentions. The crux of the technique is (1) to facilitate the modification of code that causes detected errors, (2) to prevent other codes and the entire system from being damaged despite the errors occurring in some codes, and (3) to complement the security weakness that occurs during program development or application design.

Therefore, our technique first analyzes sensitive function facets to offer a vulnerability analysis strategy for an application. Next, the technique generates a function verification operation to sanitize it. API statements that have false assertions are classified as illegal and are not allowed for execution within the application. To evaluate the proposed technique, we implemented the *Semantic Function Libraries* (SFLs) tool and tested it on a set of empirical test suites.

The main contributions of this paper are as follows:

- We describe a predictable programming technique that models semantic function verification operations for analysis.
- We introduce a new technique that estimates semantic functions by verifying input values.

The remainder of this paper is organized as follows: Section 2 reviews previous studies, and Section 3 details our research overview with a sample program, and Section 4 presents our experimental settings. Section 5 discusses our experimental findings and, finally, Section 6 presents our conclusions.

Ⅱ. Related works

2.1 Static-analysis technique

Panichella et al.[13] used the open-source checkers ITS4, Flawfinder, and RATS to analyze telecom-graded software. They found that a static code analyzer could find security vulnerabilities but produced many false negatives. In their analysis, the results were not compared with trouble reports or maintenance costs using industry data. We used a similar approach to add verification operations in the source code, but our technique is automated and does not require developer interventions such as annotations to verify an application. In addition, we do not limit our analysis to the detection of vulnerability-inducible methods. Kellogg et al.[14] proposed a lightweight bound-checking (LBC) approach. This technique uses source-to-source transformation to eliminate redundant checks. The LBC uses guard zones to separate all objects (e.g., the stack, heap, and static areas) in memory. No correct program memory can access the guard zones. However, the LBC does not handle use-after-free bugs (i.e., bugs that reference memory after it has been freed). Maletic and Collard[15] proposed srcML, which conducts static checks for custom software analysis. It consists of a representation element and toolkit for converting source code between formats. This technique marks the inherent syntax by wrapping the source code text in XML elements. The names of the elements reflect the syntax using tags such as {if}, {while}, and {class}. We use a similar approach to generate intermediate forms for our source codes.

2.2 Dynamic-analysis technique

Su et al.[16] used Fuzzy testing to challenge the reliability of a program. This testing technique detected potential vulnerabilities in binaries using test inputs to explore program control paths (the sequence of statements executed by a program) and trigger vulnerabilities. However, this technique cannot generate all possible test inputs that cover all program control paths. Therefore, vulnerabilities might be by-passed. The effectiveness of the dynamic analysis technique is based on the quality of test inputs. Without such test inputs, dynamic analysis becomes less effective in exploring application vulnerabilities. Unlike these dynamic approaches, our SFL employs a look-ahead monitoring mechanism and generates verification operations using our transformer. Thus, we guarantee that verification operations cover all control paths in a source code. Al-Shaer et al.[17] proposed LLVM, a dataflow sanitizer tool for measuring dynamic taint-tracking and secret redaction support in annotated C/C++ programs at compile-time. This technique tracks the feasibility of dynamic taint-tracking for C/C++ code that stores information in graph data structures.

III. Research overview

Fig. 1 shows a sample user-logon verification C program. This sample program consists of one Boolean variable (*PasswdStus*) and three string variables (*Passwd*, *Valid_Passwd*, and *Org_Passwd*). At line 4 in the *main*(), the *IsPassword*() is called to determine whether a user is a registered user. At line 4 in the *IsPassword*(), *gets*() function reads the logon password from the user. At line 7, the supplied password is compared with the authorized user password.

In Fig. 1, gets(Passwd) function in the IsPassword() is able to store a maximum of 15 bytes consisting of characters of the byte string identified by the Passwd variable (note that we ignore the string null terminator character). If 20-byte character strings are entered into the Passwd variable, then the Passwd variable is exploited by entering a string whose size is larger than the buffer assigned to hold it. Consequently, gets(Passwd) function may be exploited as an error or an invasion window.

Consider the following program fragment from Fig. 1.

```
 1. bool IsPassword(void) {
 2.      char Passwd [15], Valid_Passwd [15];
 3.      char Org_Passwd [20];
 4.      gets (Passwd);
 5.      …
 6.      strcpy (Valid_Passwd, Org_Passwd);
 7.      if (!strcmp (Passwd, Valid_Passwd))
 8.           return (true);
 9.      else return (false);
10. }
```

```
 1.  int main(void) {
 2.       bool PasswdStus;
 3.       puts ("Enter password:");
 4.       PasswdStus = IsPassword();
 5.       if (!PasswdStus) {
 6.            puts ("Unauthorized user");
 7.            return -1;
 8.       } else puts ("Authorized user");
 9.       return 0;
10.  }
```

Fig. 1. Simple C program

Example 1: Given the input (Valid_Passwd=NULL, Org_Passwd="EnterMyPasswordOkay"), the program generates a statement:

```
strcpy(Valid_Passwd, Org_Passwd);
```

The input is transformed into expressions:

```
strcpy(Valid_Passwd, "EnterMyPasswordOkay");
```

strcpy() function copies a maximum of 19 bytes consisting of the characters of the byte string identified by *Org_Passwd* into a character array identified by *Valid_Passwd*. *Valid_Passwd* is allocated 15 bytes by the input values. Therefore, *Valid_Passwd* is exploited.

Example 2: Continuing from Example 1, let the function for the statement be defined as follows:

```
SIZE_buf = 15;
strncpy(Valid_Passwd, Org_Passwd, SIZE_buf+5);
```

Org_Passwd is allocated 19 bytes by the input values ("EnterMyPasswordOkay"). *Valid_Passwd* is allocated 15 bytes by NULL. In this case, even though *strncpy*() function is used instead of *strcpy*() function to satisfy C secure coding rules[6], *Valid_Passwd* is exploited. Therefore, the declaration length of the variable is not an absolute evaluation element that evaluates vulnerability. Even if a semantic characteristic exists dynamically in a function, such as in Example 2, this is not detected by analyzing the static code or syntax structure[2]. Notice that string variables are compared only in terms of the structural parts of their declaration lengths. This limitation exists because static-code or syntax-aware analysis cannot detect the semantic characteristics of a function.

On the other hand, if awareness of the semantic characteristics of a function is possible, then the original code need not be modified and developed using conventional programming practices. Retrofitting an application requires manual effort proportional to the complexity of the application. Moreover, it is not necessary to hinder debugging or maintenance. To perform a semantic operation, we need to know the set of values held by functions and variables. We define a verification operation and the corresponding variable validation.

Definition 1 (Semantic verification operation associated with a control path). Let the set of function operations be denoted by *M*. For any function operation *m*, program *P* can take a semantic verification operation associated with the control path. Let *M0* denote the set of benign and safe semantic verification operations corresponding to function operation *M*[3]. Let us assume that *m* and *m0* are extracted on the same control path and that there is an associated valid function representation function Stmt_PA: $M \rightarrow M0$, $\{(m, \text{Stmt\_PA}(m)) \mid m \in \text{DOMAIN(Stmt\_PA)}\}$, and Stmt_PA $(m) = m0$. Let the set of variables (or parameters) be denoted by *V*, and for any input variable *v*, let us assume that *m* and *m0* take on the same input variable (i.e., $m \approx m0$).

Therefore, given a function *m*(*v*), if we know another valuation *m0*(*v*), then we can evaluate its verification operation and deduce the control path. Assertions have been used for performance verification with respect to partial correctness to ensure that a program does not produce unexpected results for valid operations, which are "expected"[18].

Definition 2 (Evaluation of verification assertions on a variable). For *m0* and the corresponding input variable *v*, we define assertions on variable *V*. A condition on a variable is any Boolean assertion function F: $V \rightarrow \{\text{TRUE, FALSE}\}$, $\{(v, F(v)) \mid v \in \text{DOMAIN(F)}\}$, and F(v): assert (*v.maxlen* >= *v.len*) = TRUE.

Derivation conditions of assertions are Boolean functions that are acceptable for any derivation. The role of any assertion is to indicate whether the assertion is applicable to a derivation[18]. In our approach, we can model each operation in terms of its effect on two buffer attributes: *maxlen*, the number of bytes declared for the buffer, and *len*, the number of bytes currently in use[19]. For example, after a set of string values are inputted into variables, our approach automatically calculates the length of the selected character string. This is easily implemented using the *sizeof*() function, which is defined in C functions and operations. Therefore, we can trace the buffer related variable size and status by analyzing a function's corresponding assertions. Consider the following verification assertion of *strncy*() function in Example 2. The assertion is checked as follows:

Valid_Passwd.maxlen ⊇ *min*(*Org_Passwd.len*, SIZE_buf+5)
Valid_Passwd.maxlen >= *min*(*Org_Passwd.len*, SIZE_buf+5)

Assertion checking is used to determine whether each assertion is false or checked in a program. An assertion means that *Valid_Passwd.len* should be replaced by *min*(*Org_Passwd.len*, SIZE_buf+5) and if *Valid_Passwd.maxlen* < *min* (*Org_Passwd.len*, SIZE_buf+5) is TRUE, then this is a fault. Therefore, given input variables in a program, a potential exploitation can be considered safe if all function assertions are TRUE[20].

IV. Implementation

4.1 The tool: SFL

Our tool consists of three components: (1) preprocessing, (2) source code transformation, and (3) source code sanitization. Preprocessing is implemented using a source code facet analysis technique. The source code transformation is implemented using a C/C++ source transformation tool. Finally, the source code sanitization is implemented using our Pattern Analysis (PA) library.

4.1.1 Preprocessing: Source code function facet analysis

We preprocess the source programs text segmentation and stop words list. In order to extract complete command keywords, we remove comments and external libraries. For vulnerability command extraction, we utilize suffix array based technique to extract vulnerability commands, and then select the keywords from them with rules. Specifically, we use the phrase discovery algorithm [21], which employs a variant of suffix arrays extended with an auxiliary data structure. Then, we remove the single words that are comments and the phrases which begin or end with stop words. We inquire into the words that are single words and the phrases. Finally, the filtered words and phrases that exceed the com-mand keyword frequency threshold *T* are chosen as command keywords. Additionally, removing the safe commands can reduce the calculating complexity of the method. These results are used to configure the indexed fingerprint database (see Section 4.1.2).

Since programs are composed of a set of functions, if a vulnerable function exists, the application may be vulnerable. Sensitive function analysis of source code is significant because vulnerability-inducible functions cause potential faults or bugs in a program. Therefore, function facet analysis of source code can offer a vulnerability analysis strategy for an application. We define function keyword to measure function facets. Fig. 2 shows an example of a function text enumerator based on the coverage of function keyword in C/C++.

```
Enum Method Types
    None = 0;
    gets = 1;
    strcpy = 2;
    strcat = 3;
    ...
    fgets = 23;
    strncpy = 24;
    strncat = 25;
End enum
```

Fig. 2. Example of function type statements

Table 1. Summary of notations

Notations	Description
FF(kw)	The function facet of keyword(kw)
FK(kw_j,s_i)	The frequency of keyword(kw) in source code(s)
t(kw_j,s_i)	The number of times that keyword(kw) appears in source code(s)

We assume that we can extract a set *KW* of *q* keyword from source codes *S*. Therefore, we define the function facet (*FF*) set as fllows:

$$FF(kw) = FF(kw) \cup \sum_{i=1}^{n}\sum_{j=1}^{q} FK(kw_j, s_i) \tag{1}$$

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

$$FK(kw_j, s_i) = \sum_{i=1}^{n}\sum_{j=1}^{q} t(kw_j, s_i) \tag{2}$$

where $t(kw_j,s_i)$ represents the number of times that keyword kw_j appears in source code s_i. During the facet analysis process, the function keyword is classified as a weakness as follows, where the vulnerability-inducible function keyword set KW_c contains the keyword covered by the vulnerability-inducible function:

$$t(kw_j, s_i) = \sum_{i=1}^{n}\sum_{j=1}^{q} (kw_j, s_i, KW_c) \tag{3}$$

The preprocessing working scheme is described briefly in Fig. 3. At the beginning of preprocessing, no source code is selected (*SP* = 0). λ ($\lambda \in [0,1]$) is used to give a tradeoff score between a vulnerable function and a safe function. At line 2, the procedure reads the first source code s^*[22], and then the set KW_{s^*} of keywords that s^* selects is compared with the vulnerability-inducible function keyword set KW_c. Next, the facet relevance score of the vulnerability- inducible function is calculated.

```
Preprocessing ($S$, $KW$, $\lambda$)
 1.  Initial $SP = 0$
 2.  Read the first source code $s^*$ ($s^* \in S$)
 3.  Loop
 4.      $KW = KW_c \cap KW_{s^*}$
 5.      If $KW \subseteq KW_c$ then
 6.          $SP = SP \cup \{ s^* \}$
 7.          $S^* = (1-\lambda)t(kw,s), \forall s \in S$
 8.      End if
 9.      If $EOF$ then
10.          Exit loop
11.      Else
12.          Read the next source code $s^*$ ($s^* \in S$)
13.      End if
14.  End Loop
15.  return $SP$
```

Fig. 3. Preprocessing working scheme

Specifically, since source codes have many different facets of vulnerability-inducible functions, we suppose that a set of keyword exists whose different subsets represent the various underlying facets well. We can then cover all the facets by selecting a source code subset to cover the keyword. The procedure is repeated until all function keyword are analyzed. Vulnerability scores indicate the riskiness of the vulnerable function. Vulnerability scores range over 0 - 1, with 0 representing no vulnerability and 1 representing high vulnerability.

4.1.2 Tree-based source code transformation

A source code transformation using Abstract Syntax Tree (AST) is the logical approach for analyzing an application written in C/C++[23]. We implemented index fingerprint representations using a robust parser [24] capable of extracting syntax trees from source code even without a working build environment. In particular, this parser does not validate the syntax of the source code. Instead, the aim is to extract as much information from the source code as possible. The AST subtrees correspond to an executable statement or control-flow structure in the source code. Each subtree can be represented as a fingerprint set with fingerprint indexation in a customized database for further reference (e.g., match back-tracking). The fingerprint database is indexed. Fingerprints are sorted by decreasing weight, hash value, and parent-linked nodes. This structure can be used to iterate over the database to retrieve function calls and obtain all the fingerprints of the child subtrees of a given node. That is, the group of each subtree corresponding to an instruction statement of exact subtree matches can be retrieved by iteration over the indexed fingerprint database.

To perform source code transformation, we used the TXL source-code transformation algorithm[25]. The TXL algorithm consists of two phases. The first is the use of a grammar file to produce a scanner and a parser for the grammar. In this phase, our AST is constructed from a robust parser and is used as the input. The second phase applies the transformation rule to the subtree. This specifies any sequence of external definitions such as function definitions or global declarations (e.g., memset, global_fun). In our approach, we preprocessed the source code analysis to extract function facets using a function detection enumerator. If we only consider a particular function facet, our experiment is half-learned and checks the shallow functionality of the program. Therefore, we can establish a source code transformation strategy to analyze a particular function facet.

We add a function verification operation to instrument a set of variables. For example, if x and y are two input variables in *strcpy*() function, the operation:

strcpy (x, y);

results in a verification operation. The transformer adds a verification operation prior to this operation as follows:

Stmt_PA (strcpy, x, y, sep);

(String=*Stmt_PA*(function_pattern, var*1*,⋯, var*n*, [separator];)

Notice that this function has the original input variables and can therefore perform real computation along the actual control path taken by the program. The conditional expression in the original source code is snot modified by our transformer. Fig. 4 shows the partial transformed code for the program in Fig. 1.

```
bool IsPassword(void) {
    …
    Stmt_PA(strcpy, Valid_Passwd, Org_Passwd, 1);
    strcpy (Valid_Passwd, Org_Passwd);
    …
    Stmt_PA(strcmp, Passwd, Valid_Passwd, 2);
    if (!strcmp (Passwd, Valid_Passwd))
    …
}
```

Fig. 4. Practice example program

4.1.3 Semantic-code based sanitization

Sanitization is a specific process type of input validation performed before external inputs are used. To guarantee quality assurance of the variable and verify the function operation, we implemented a PA library that represented the verification pattern information of the function in the program control path. PA is based on function assertions, which are verification conditions wherein a predicate should always be true at that point in the code. At runtime, if an assertion evaluates as false, it causes an assertion failure. The program then crashes or produces an assertion exception.

Fig. 5 shows a heuristic function assertion example in the PA library for *Stmt_PA(strncpy*, *dst*, *src*, *n*). A verification operation is performed to check function assertion according to the pattern information in PA.

Method assertion

```
/* +1 is for adding the null terminator */
if (dst.maxlen >= src.len)
    strncpy(dst.val, src.val, n);
else if (dst.maxlen < src.len && dst.maxlen >= n + 1)
    strncpy(dst.val, src.val, n);
else/* Overflow */
    print ("Buffer overflow %d %s\n", src.len, src.val);
```

Fig. 5. An example of a string verification operation

PAs are connected by the *Stmt_PA*(). *Stmt_PA*() calls a PA that verifies the function variable for vulnerability detection. This operation produces an exception if the PA returns false. Otherwise, the statement may cause vulnerability at runtime. Before a statement is executed, PA pre-examines the variable for vulnerability. This idea of verifying variables in the program control path differs from the traditional dynamic tainting analysis, in which inspection is done after execution. Thus, we can monitor function variables before the statement is executed in an actual control path. PA has two major steps: step 1 analyzes the grammatical patterns of the statements and then computes and stores all variable attributes (i.e., maxlen and len). Step 2 computes variable assertions to classify vulnerability.

```
Verify_Substitute_Stmt (PA_patterns: pattern information; V1, ... , Vn: variables
                                  | N | : number of variables)
 1.  Initial m = 0, i = 1, k = 1, p = 1
 2.  Read pattern_operation_stmt
 3.  m = |N|                          {current number of variables}
 4.  For i = 1 to m
 5.      Read_pattern[i] = Read_params[i]
 6.      Patseti[] = AnalyzePattern (Read_pattern[i])
 7.  Next i
 8.  For i = 1 to i(max)
 9.      If (ComputePatSet (Patseti []) <> NULL) Then
10.          PSstmt [k] = ComputePatseti[]     {variable assertion}
11.          Increment k
12.      End if
13.  Next i
14.  ComparePatternValues (PSstmt [p])
End Procedure
```

Fig. 6. A brief outline of PA-based estimation

Fig. 6 shows the scheme in brief for verification operations. PA calls the *Verify_Substitute_Stmt*() procedure, which verifies the input variables in accordance with the verification pattern information. This procedure produces an exception if there is no assurance in the results. The *AnalyzePattern*() analyzes functions and calculates function variable attributes. Function variable assertions, calculated by the *ComputePatSet*(), are stored in *PSstmt*[], and each *PSstmt* is compared using the *ComparePatternValues*().

V. Evaluation

5.1 Empirical test-suite evaluation

We evaluated the detection accuracy using the Juliet test set[26] to assess the effectiveness of the *SFL*. This test set is a collection of C/C++ programs with known flaws consisting of 23,957 test cases documented by Common Weakness Enumeration (CWE)[27]. It has been used to understand other software assurance tools' capabilities[28]. In addition, it covers the top 25 security errors defined by SANS/MITRE (The MITRE Corporation, 2017, http://cwe.mitre.org/). The CWE entries were composed of one or more abstract categories. For example, the "Buffer overflow" category is represented by different CWE entries (e.g., CWE-131: incorrect calculation of buffer size; CWE-191: integer overflow).

Therefore, we modeled CWE entries according to the "Center for Assured Software 2016" to generate a general software security valuation model[29]. This security model helps to analyze and interpret the Juliet test set results. For accurate analysis, we classified every case into two cases. We also removed some conditional preprocessor (*#ifndef*) commands. The test set was classified into two types of test cases: flawed or "*bad*" code and "*good*" code, which contains legitimate inputs. The "*bad*" code is used to check whether the proposed *SFL* successfully detects flaws and reports false negatives. On the other hand, the "*good*" code is used to verify whether any false positives are reported. If the *SFL* modifies input, we obtain a false positive.

We performed an experiment to determine whether false positives or false negatives were caused by a loss of precision. We ran the test set to ensure that the *SFL* successfully detected and prevented flaws. The flaws suite includes more than 10 different kinds of flaws such as buffer overflow flaws and code quality flaws. The results of running the test cases are summarized in Table 2.

Table 2. Juliet test set evaluation results.

Weakness type	Test CWEs	SLOC	Input Attempts		Cases/ Flaws	False Negatives (%)	Arbitration
			Good Code	Bad Code			
Memory corruption	CWE-120	2,210	8	61	84/84	10.7	9
	CWE-126	2,852	6	55	106/106	0	0
	CWE-131	2,032	4	42	51/51	1.9	1
	CWE-190	3,116	3	67	86/86	0	0
	CWE-191	2,191	2	33	38/38	18.4	7

The first section lists the weakness type of the security model. “Memory corruption” has the most CWE entries (buffer handling) because this is the most frequent type of security vulnerability in C/C++ programs. The third section lists the size of the Source Line of Code (SLOC). The fourth section lists the number of test cases of the two types. The fifth section shows the number of cases in the test sets and the number of flaws detected by the *SFL*. The sixth section shows the rate of false negatives, as there is a situation in which the proposed *SFL* fails. The final section shows the number of developer interventions (e.g., parse errors).

5.2 Performance accuracy

Our proposed technique was compared with three existing well-known detection approaches: Flawfinder (static analysis; Ver. 1.31), Cppcheck (static analysis; Ver. 1.76), and Visual Studio Compiler (static and dynamic analysis; Ver. 2015). The accuracy of vulnerability detection indicates the number of exactly detected flaws. We compared the detection accuracy and the time complexity imposed by the approaches. The applications were installed on the local host to prevent network overhead. We performed test suite runs and measured the detection accuracy of each run with caching disabled. For reasons of accuracy, we performed our experiments three times and report the average run time. The Flawfinder tool detects only fixed-size buffer overflow flaws, such as exceeding the buffer size. The Cppcheck tool can detect buffer overflow and format string flaws. Visual Studio Compiler found by far the most flaws.

However, most of them were non-security issues. Furthermore, it took the longest time to analyze our test set. On the other hand, our *SFL* tool is effective in detecting and preventing flaws. In addition, the *SFL* tool yields 18–23% better accuracy than the Flawfinder and Cppcheck tools.

In addition to detection time, time complexity is an important factors for evaluating a detection system. Table 3 compares the algorithms with respect to deployment requirements. The term 'High' has twice the time complexity of the term 'Medium'. The term 'Very high' has twice the time complexity of the term 'High'.

Table 3. Comparison of performance between algorithms

Algorithm	Detection	Prevention	Time complexity
Flawfinder	Yes	No	Low
Cppcheck	Yes	No	Low
Visual studio compiler	Yes	No	Very high
SFL	Yes	Yes	Very high

VI. Conclusion and future work

In this paper, we have proposed a technique that detects and prevents vulnerabilities in an application. In applications, most vulnerabilities stem from numerous unexpected input values. However, retrofitting an application requires significant resources. It is therefore necessary to predict and trace the locations of bugs throughout the course of a program's operation. It is essential to detect errors and to minimize damage when uncontrollable or unexpected errors occur. Our approach provides a technique for validating and sanitizing instrumented code through a semantic function evaluation technique from input data in C/C++ programs, which are widely used for critical software. The proposed method is based not only on information needed to check flaws in a program, but also on the verification of statements using semantic code evaluation. The results provide evidence that it is possible to verify statements prior to their execution, thereby preventing malicious attacks. Our approach is differentiated from previous published techniques in the following ways:

(1) We attempted to verify code vulnerabilities by generating function sanitization.
(2) Our approach is based on semantic verification operations, but differs from traditional dynamic tainting analysis, which inspects results only after the execution of a program.

We have developed a novel technique which is useful in enforcing software security monitoring. However, further studies must be conducted on sanitization quality activities to enhance our approach.

References

[1] G. Fengjuan, W. Linzhang, and L. Xuandong, "BovInspector: automatic inspection and repair of buffer overflow vulnerabilities", 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, Singapore, pp. 786-791, Sep. 2016.

[2] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs", Journal of Systems and Software, Vol. 117, pp. 166-184, Mar. 2016.

[3] P. Bisht and P. Madhusudan, V. N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks", ACM Transactions on Information and System Security, Vol. 13, No. 2, Article No. 14, Feb. 2010. doi>10.1145/1698750.1698754.

[4] A. Mahmoud and G. Bradshaw, "Semantic topic models for source code analysis", Empirical Software Engineering, Vol. 22, No. 4, pp. 1965-2000, Aug. 2017.

[5] A. Denault, "Defensive programming introduce to software system Lecture 18", Computer science, McGill University, pp. 58-72, 2015.

[6] Seacord R. C, "The CERT C Secure Coding Standard(2nd Edition)", Boston: Addison-Wesley, pp. 121-153, 2016.

[7] F. Hujainah, R. Bakar, M. Abdulgabber, and K. Zamli, "Software Requirements Prioritisation: A Systematic Literature Review on Significance, Stakeholders, Techniques and Challenges", IEEE Access, Vol. 6, pp. 71497-71523, Nov. 2018.

[8] T. Ye, Z. Lingming, W. Linzhang, and L. Xuandong, "An empirical study on detecting and fixing buffer overflow bugs", 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, pp. 91-101, Apr. 2016.

[9] J. Maletic and M. Collard, "Exploration, analysis, and manipulation of source code using srcML", IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, Vol. 2, pp. 951-952, May 2015.

[10] A. Wagner and J. Sametinger, "Using the Juliet Test Suite to Compare Static Security Scanners", 11th International Conference on Security and Cryptography (SECRYPT), Vienna, Austria, pp. 244-252, Aug. 2014.

[11] Y. S Jang and J. Y Choi, "Detecting SQL injection attacks using query result size", Computers & Security, Vol. 44, pp. 104-118, Jul. 2014.

[12] S. Hossain, M. Hisham, and V. Ishan, "Buffer overflow patching for C and C++ programs: rule-based approach", ACM SIGAPP Applied

Computing Review, Vol. 13, No. 2, pp. 8-19, Jun. 2013.

[13] S. Panichella, V. Arnaoudova, M. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews", 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, pp. 161-170, Mar. 2015.

[14] M. Kellogg, V. Dort, S. Millstein, and M. Ernst, "Lightweight verification of array indexing", In 2018 International Symposium on Software Testing and Analysis, Amsterdam, Netherlands, pp. 3-14, Jul. 2018.

[15] J. Maletic and M. Collard, "Exploration, analysis, and manipulation of source code using srcML", 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, pp. 951-952, May 2015.

[16] S. Su, M. Chen, and Y. Hsueh, "A novel fuzzy modeling structure-decomposed fuzzy system", IEEE Transactions on Systems, Vol. 47, No. 8, pp. 2311-2317, Aug. 2017.

[17] E. Al-Shaer, J. Wei, K. Hamlen, and C. Wang, "Deception-enhanced threat sensing for resilient intrusion detection", In Autonomous Cyber Deception, pp. 147-165, Jan. 2019.

[18] L. Peng and L. Wise-Faberowski, "An unexpected case of post-operative superior caval vein syndrome", Cardiology in the Young, Vol. 28, No. 6, pp. 879-881, Jun. 2018.

[19] J. Wu, Y. Wang, P. Wang, J. Pei, and W. Wang, "Finding Maximal Significant Linear Representation between Long Time Series", In 2018 IEEE International Conference on Data Mining (ICDM) Singapore, Singapore, pp. 1320-1325, Nov. 2018.

[20] S. Kiebzak, G. Rafert, and C. E. Tucker, "The effect of patent litigation and patent assertion entities on entrepreneurial activity", Research Policy, Vol. 45, No. 1, pp. 218-231, Feb. 2016.

[21] F. A. Louza, S. Gog, and G. P. Telles, "Inducing enhanced suffix arrays for string collections", Theoretical Computer Science, Vol. 678, pp. 22-39, May 2017.

[22] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan, "Auditing buffer overflow vulnerabilities using hybrid static dynamic analysis", IET Software, Vol. 10, No. 2, pp. 54-61, Apr. 2016.

[23] Y. Sun and J. Gray, "A demonstration-based model transformation approach to automate model scalability", Software & Systems Modeling, Vol. 14, No. 3, pp. 1245-1271, Jul. 2015.

[24] C. M. Agulhari, A. Felipe, R. C. Oliveira, and P. L. Peres, "Manual of the Robust LMI Parser", Version 3.0, Oct. 2018.

[25] V. Martínez, M. S. Serpa, P. J. Pavan, E. L. Padoin, and P. O. Navaux, "Performance Evaluation of Stencil Computations Based on Source-to-Source Transformations", Latin American High Performance Computing Conference, Bucaramanga, Colombia, pp. 213-223, Sep. 2018.

[26] NIST, Software Assurance Reference Dataset, http://samate.nist.gov/SRD/testsuite.php. [accessed: Aug. 21, 2019]

[27] CWE, CWE – Common Weakness Enumeration, http://cwe.mitre.org, [accessed: Aug. 21, 2019]

[28] A. Nanthaamornphong and J. C. Carver, "Test-Driven Development in scientific software: a survey", Software Quality Journal, Vol. 25, No. 2, pp. 343-372, Jun. 2017.

[29] NSA, NSA Center for Assured Software, http://cps-vo.org/node/1529. [accessed: Aug. 21, 2019]

[30] D. J. Jeon and D. G. Park, "Real-time linux malw are detection using machine learning", Journal of KIIT, Vol. 17, No. 7, pp. 111-122, Jul. 2019.

[31] H. K. Chin and J. H. Ahn, "Duplicated control dat a purging algorithms for SBML protocol tolerating temporary communication errors", Journal of KIIT, Vol. 17, No. 5, pp. 21-27, May 2019.

Author

Young-Su Jang

2011 : M.S degree in Department of Computer Science from Korea University

2019 : PhD degrees in Department of Computer Science from Korea University

2017. 10 ~ present : Assistant professor with the smart software department, Korea Polytechnic

Research interests : secure software engineering, secure coding, and formal method