PAPER

Automatic Prevention of Buffer Overflow Vulnerability Using Candidate Code Generation

Young-Su JANG^{†a)}, *Member* and Jin-Young CHOI^{††b)}, *Nonmember*

SUMMARY The security of a software program critically depends on the prevention of vulnerabilities in the source code; however, conventional computer programs lack the ability to identify vulnerable code in another program. Our research was aimed at developing a technique capable of generating substitution code for the detection of buffer overflow vulnerability in C/C++ programs. The technique automatically verifies and sanitizes code instrumentation by comparing the result of each candidate variable with that expected from the input data. Our results showed that statements containing buffer overflow vulnerabilities could be detected and prevented by using a substitution variable and by sanitizing code vulnerabilities based on the size of the variables. Thus, faults can be detected prior to execution of the statement, preventing malicious access. Our approach is particularly useful for enhancing software security monitoring, and for designing retrofitting techniques in applications.

key words: *information security, buffer overflow vulnerability, software security monitoring*

1. Introduction

The quality of software code is closely associated with a variety of factors, such as the security and maintainability of a program, and a reduction in quality is closely associated with most program errors arising from unexpected input values [1], [2]. It has also been presumed that the safety of software would be increased if software code was error free, which is essential for preventing damage to the system [3]. However, this is not applicable to all cases [4], [5]. One reason is that even a code analysis tool cannot detect all the vulnerabilities that may be present in a program [6]. Another reason is that false positives (hereinafter called “F/P”), which are legitimate inputs identified as vulnerabilities, may occur with model checking-based analysis because of incomplete or incorrect legitimate inputs [7]. Moreover, sanitization analysis of regular expressions and built-in validation functions is ineffective in detecting vulnerabilities in input sources arising from either a user or another program [8]. Structural matching, such as secure code, is also vulnerable to F/P when a statement is transmitted to the expected structure [9].

Both static and dynamic code analyses are well-known approaches that can be used to ensure the quality of software. Static analysis is performed to assess all possible

pathways extending from code checking to latently vulnerable statements. However, a static approach may produce F/P during the analysis of complex programming constructs or non-trivial functions intended for sanitization [10]. On the other hand, although dynamic code analysis can accurately detect vulnerabilities, identifying paths associated with the detected vulnerabilities is difficult using this method [1].

We propose a technique that generates substitution code for the detection and correction of buffer overflow vulnerabilities in C/C++ applications. We call its implementation as Vulnerability Detection Libraries (VDLs). The proposed technique automatically verifies whether a statement is transmitted at each location in the program. Moreover, it performs an analysis of the information that is needed to check flaws in statements and an analysis of these statements through evaluation of the substitution code. This method is based on additional analysis to determine whether the substitution code follows the intended results of the statement. Therefore, our technique first infers the size of variables and stores this information in new substitute variables. Next, an instrument is used to check substitute variables before sensitive operations, and this occurs at runtime. Thus, vulnerable statements that produce results are classified as illegal and then not permitted for execution of the program. In the current study, we evaluated whether our proposed technique is useful in detecting buffer overflow flaws in an empirical application setting. The paper makes the following contributions:

- (1) A substitution code evaluation technique that models how an application processes the sanitization of candidate variables.
- (2) An attempt to ascertain the correctness of the sanitization process and thereby identify buffer overflow vulnerabilities by using a substitution code generation technique.

The paper is structured as follows: Sect. 2 reviews previously reported studies. Section 3 discusses our research overview with a sample C source program, and Sect. 4 illustrates our approach using an example. Section 5 details the implementation, Sect. 6 contains a comparative evaluation, and Sect. 7 presents our conclusions.

2. Literature Review

In the current study, we classify previously published studies into four groups: (1) static code analysis; (2) dynamic code

Manuscript received May 29, 2018.

Manuscript revised July 24, 2018.

Manuscript publicized August 24, 2018.

[†]The author is with Korea Polytechnics, Ansong, Korea.

^{††}The author is with Korea University, Seoul, Korea.

a) E-mail: jyskkh@kopo.ac.kr

b) E-mail: choi@formal.korea.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2018EDP7192

analysis; (3) model checking based analysis; (4) concolic testing.

Static code analysis. Previous studies examined the level of maturity of software development using source code scanning tools, thus confirming that a static code analyzer is useful for detecting security vulnerabilities [11]. Flawfinder, a source code scanning tool, provides a database of Application Programming Interface (API) symbols that are commonly seen in association with the vulnerabilities. This database can be used for evaluating capability of vulnerability detection through static code analysis [12]. We use a similar approach to define an authorized function text value (see Sect. 5.1.2).

Dynamic code analysis. The DynaMine tool is used to analyze the revision histories of source code check-ins and thereby to identify correlated method calls and common error corrections to find coding patterns that are application specific [13]. It combines the revision history mining with techniques that allow dynamic analysis. It is therefore effective for the discovery of new patterns that are application specific as well as for detecting errors in extremely large-scale applications. Fuzz testing [14], [15] is traditionally used to challenge the reliability of an application with sensible test inputs. This technique can detect latent vulnerabilities in applications because the test inputs might explore program control paths and trigger the vulnerabilities. However, this technique cannot generate all possible test inputs that cover all program control paths. Source code assertions, also known as legality assertions, are inserted prior to dereferencing each subscript and pointer to confirm whether the expression used for referencing indicates a location inside the object or an array that is identified during runtime [7], [16].

Model checking based analysis. CCured is a tool capable of transforming any C program to a version that is memory-safe [17]. It can verify memory type-safety by using an additional runtime check to detect and verify safe pointer usage. The tool can transform a legacy C program into one that is memory-safe by converting most of its pointers into fat pointers. A string analysis-based framework that generates vulnerability signatures in the presence of bug patterns such as regular expressions has been developed [18]. Forward and backward symbolic reachability analyses were employed to construct these signatures, both of which provide over-approximations. The type-safe variant Cyclone proposes safe dialects for C [19]. This prevents more attacks than data-flow integrity enforcement because it ensures memory safety for programs written in these dialects. However, it requires considerable effort to port applications coded in C to safe dialects. In addition, major runtime changes are required. For instance, the tool replaces *malloc* and *free* with a garbage collector, which complicates performance prediction and can introduce significant overhead.

Concolic testing. SWI-Prolog is used for concolic testing associated with optional generation of a logic program [20]. It performs experimental coverage analysis to compute the percentage of used clauses and failing clauses

using key-value sets as primary elements. However, it requires developer intervention to specify filtering rules for each application input. On the other hand, our technique is automated and does not require developer intervention, such as filtering rules, in order to detect buffer overflow vulnerabilities. Symbolic execution semantics, called concolic execution, for logic programs has been presented [21]. The technique extends concrete execution by allowing symbolic input data and exploring all feasible execution paths. We use a similar approach to define a symbolic expression. However, this approach only considers simpler statement coverage. The differences to our work are twofold. First, our technique is to verify the correctness of the sanitization process using a substitution code generation technique and do not limit our analysis to detecting of buffer overflow vulnerabilities. Second, our technique employs an additional dynamic analysis to detect buffer overflow vulnerabilities.

3. Research Overview

3.1 Typical Example

We illustrate the vulnerability detection process by considering the sample C source program given in Fig. 1. It is a simple string length comparison program consisting of one file open pointer variable (*pFile*) and three string variables (*pBuffer*, *cmp_buffer*, *max_buffer*). At line 7, the *fgets()* function reads the string from an external file. At line 8, the *strncpy()* function copies a maximum of *STR_SIZE + 15* (30 bytes) characters from the byte string identified by *pBuffer* to the character array identified by *cmp_buffer*. At lines 9 and 10, after comparing the character string length of the *cmp_buffer* and *max_buffer*, the maximum length of the character string is stored in the *max_buffer*.

3.2 Research Motivation

For every variable in the source code, all available values at each statement can be computed and then stored. In C, arrays are often used by developers to compute and store values for reuse in the next step. Arrays are used everywhere

```

1. int main(void) {
2.     #define STR_SIZE 15
3.     FILE *pFile;
4.     char pBuffer[STR_SIZE+15];
5.     char cmp_buffer[STR_SIZE];
6.     char max_buffer[STR_SIZE];
7.     ...
8.     while (fgets(pBuffer, STR_SIZE+15, pFile) != NULL) {
9.         strncpy(cmp_buffer, pBuffer, STR_SIZE+15);
10.        if (strlen(cmp_buffer) > strlen(max_buffer))
11.            strncpy(max_buffer, cmp_buffer, STR_SIZE);
12.    }
13.    return 0;
14. }
```

Fig. 1 Sample C source program.

because they are convenient and can be allocated and deallocated without developer intervention. However, in spite of their convenience, stack buffer overflows are the most common security vulnerabilities in C programs [22]. Consider the following program fragment from Fig. 1.

Example 1: Given the input (*pBuffer* = “Green Eggs and Ham”), the program executes and generates a statement:

```
strncpy(cmp_buffer, pBuffer, STR_SIZE + 15);
```

Transforming the input into the expressions:

```
strncpy(cmp_buffer, “Green Eggs and Ham”, 30);
```

In the above statement, the `strncpy()` function copies a maximum of 30-byte characters from the byte string identified by *pBuffer* to a character array identified by *cmp_buffer*. The *pBuffer* variable is allocated 18 bytes in accordance with the input values (“Green Eggs and Ham”). Thus, `sizeof(pBuffer)` is 18. On the other hand, the *cmp_buffer* is allocated 15 bytes. Consequently, a typical way to exploit the *cmp_buffer* is to use as input a string with a size that exceeds that of the buffer assigned to contain it. That is, unlike the requested length of a variable in the source code, the variable only takes the declaration length into consideration. Analysis of the static code or statement syntax structure (e.g., syntax tree) would not indicate the existence of an anomaly in the statement. This is because static code or syntax-aware analysis cannot detect dynamic semantic statements. We therefore inevitably need to check the length of the variables before the statement executes. However, retrospectively enabling an application to employ the `strlen()` function would require manual specification of the intended statement at every point. In this case, the required effort would be in proportion to the complexity of the application.

3.3 Our Approach

Our main research idea involves verification of the substitution source code for the developer-intended statement whenever the *control path of the program* (the sequence of statements the program executes) reaches the location of a program. During execution, the intended statement is presented as a set of program inputs and string constants generated by the program [23]. Considering such inputs and constants, we detect and prevent faults by comparing a candidate variable from the intended statement with the possible faulty statement. Our approach essentially ensures the sanitization of candidate variables to identify latent vulnerabilities that may bypass the sanitization and vulnerable elements in the verification process.

A candidate variable can be any arbitrary C expression that resolves to a scalar type. However, a candidate variable is required to be an *lvalue* [24]. We can declare a candidate variable according to the rules for C. Table 1 shows the false and true types of C data we use in declaring a candidate variable. For example, we can use the special REPresentative CHARACTER (REP_CHAR) pseudo-type to declare character strings. REP_CHAR is an extended C type or predeclared

Table 1 C data types of candidate variables.

C data type or pseudo type	Description
REP_CHAR (= char)	Single character
REP_INT (= int)	Integer
REP_SHORT (= short)	Small integer
REP_LONG (= long)	Large integer
REP_FLOAT (= float)	Floating-point number (single precision)
REP_DOUBLE (= double)	Floating-point number (double precision)
REP_CHAR[n]/[m][n]	A substitute[n]/[m][n] character array (string)

C *struct* and the preprocessor expands REP_CHAR into the following *struct* with value and length members:

```
unsigned char  val [n];    /* Value    */
unsigned short len;       /* Length  */
unsigned short maxlen;    /* Maximum length */
```

The value, length, and maximum length members hold the current string value, the current length of the value stored in the value member, and the declaration length of the variable, respectively. The advantage of using REP_CHAR variables is that we can explicitly refer to the length member of the REP_CHAR structure. After a set of values is input into REP_CHAR variables, our VDL automatically inserts the length of the selected character string into the length member. This is easily implemented using the unary operator `sizeof`, which generates the size of a variable or datatype.

Example 2: Consider the following string concatenate `strcat()` function: Let *uname* and *emp_name* be the two variables in the operation, where *uname* is a pointer to the null-terminated byte string to copy from, and *emp_name* is a pointer to the byte string to copy to:

```
1. REP_CHAR  uname [10];
2. REP_CHAR  emp_name [20];
3. gets(emp_name.val);
4. strcat(uname.val, emp_name.val);
```

Let us assume that a REP_CHAR *uname* is initialized by NULL and the *emp_name* input value is as follows:

```
emp_name.val = “Harry Potter”
```

In this case, *emp_name.len* (12 bytes) is greater than *uname.maxlen* (10 bytes). Thus, a buffer overflow must occur. On the other hand, consider the input value for the variable as follows:

```
emp_name.val = “Peter Pan”
```

In this case, *emp_name.len* (9 bytes) is less than *uname.maxlen* (10 bytes). Thus, a byte string identified by *emp_name.val* is appended to a byte string identified by *uname.val*. Consequently, our VDL compares the actual

length of the character string stored in the “*val*” member before the statement is executed in the source code.

In our study, we aimed to predict the intended results of the statement by executing the application. The following conditions should be met for the generation of the substitution source code:

- (1) It is mandatory to add candidate variables, denoting its applicant, to variables and parameters in the source code.
- (2) It is mandatory to ensure the security of the value of variables in the substitution source code. Thus, the resulting statement cannot contain faults.
- (3) It is mandatory to execute the value of candidate variables in a program control path by considering the value of real variables in the source code.

4. Formal Analysis Using Symbolic Expressions

This section formalizes candidate code generation. When actual input is used to run a program, it follows a control path and the statements constructed along this path can be considered a symbolic expression with parameters created based on the input variables [21]. An intuitive method to verify the intended statement involves substituting benign candidate inputs in the symbolic expression of the statement. A set of possible values for a program variable is needed to verify the sanitization of candidate variables associated with vulnerable code statements. This led us to define symbolic execution as a technique that is commonly used to verify programs [25]. That is, a symbolic expression is defined as a statement constructed on a specific path and then parameterized on input variables when a program is run using actual input. Statements generated by a program can

be represented as symbolic expressions relating to a set of program inputs.

We use simple programming to enable us to concentrate on the main ideas of our research. Then, we define simple *if-else programming* with two variable types: numbers and strings. Further, we specify two sets of variables: numerical variables N and string variables S , with n denoting a number variable and s denoting a string variable. A set of functions f_i , each of which accepts as input a tuple of number/string values and then returns a number, is also specified. In addition, we define a set of functions g_i that accepts a tuple of string/number values and then produces a string. The *verify_stmt()* function is used as a verification operation. We make the assumption that a non-input number variable is initially assigned the value of zero and that a non-input string is initialized to NULL. Figure 2 presents the syntax of the programs. This programming language is similar to the simple programming language utilized in [26]. For notational simplicity, we use the notation “|”, which can be interpreted as “or,” to signify multiple rewriting rules contained in a single line.

4.1 Candidate Variable Definition and Verification

Figure 3 shows a partial program for practical illustration purposes. The program contains the number n and the strings *org_buffer* and *dest_buffer* as input variables and REP_CHAR data type *REP_org_buffer* and *REP_dest_buffer* (candidate variables of *org_buffer* and *dest_buffer*, respectively) as substitute ones. There are two blocks of statements. In particular, different statements are generated by the program depending on the value of n . The value used as input for n determines the control path followed by the program. That is, program P follows a unique control path

$P := \text{defn}; \text{stmt};$	(program)
$\text{verify_stmt}(\text{string}_1, \dots, \text{string}_k, \text{number}_1, \dots, \text{number}_k)$	
$\text{defn} := \text{int } n \mid \text{char } s \mid \text{\#define cost} \mid \text{defn}$	(variable declaration)
$\text{stmt} := \text{if } (\text{exp}) \{ \text{stmt} \} \text{ else } \{ \text{stmt} \} \mid \text{stmt}$	(statement)
$\mid n := ne \mid s := se \mid \text{term assop term} \mid \text{skip}$	
$\text{exp} := \text{term relop term} \mid \text{term}$	(expression)
$\text{term} := \text{id} \mid \text{sign digit}$	(term)
$\text{assop} := = \mid + = \mid - = \mid / = \mid * =$	(assignment operators)
$\text{relop} := < \mid < = \mid == \mid != \mid > = \mid >$	(relational operators)
$ne := n \mid f_i(t_1, \dots, t_k)$	(arithmetic expressions)
$se := s \mid g_i(t_1, \dots, t_k)$	(string expressions)
$\text{id} := \text{letter} (\text{letter} \mid \text{digit})^*$	(identifier)
$\text{letter} := [\text{A-Za-z}]$	(letter)
$\text{sign} := + \mid -$	(sign)
$\text{digit} := 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	(digit)
where $n \in N$ is any number constant,	
$s \in S$ is any string constant,	
each t_i is either ne or se , depending on the parameters for f_i, g_i , respectively.	

Fig. 2 Simple if-else programming.


```

Program P
1. #define STR_SIZE 10
2. input  int n;
3. input  char  org_buffer[STR_SIZE];
4. input  char  dest_buffer[STR_SIZE+10];
5. REP_CHAR  REP_org_buffer[STR_SIZE];
6. REP_CHAR  REP_dest_buffer[STR_SIZE+10];
7. if (n == 0) {
8.     verify_stmt(strncpy, REP_dest_buffer,
9.                 REP_org_buffer, STR_SIZE);
10.    strncpy(dest_buffer, org_buffer, STR_SIZE);
11. }
12. else {
13.     verify_stmt(strncpy, REP_org_buffer,
14.                 REP_dest_buffer, STR_SIZE+10);
15.    strncpy(org_buffer, dest_buffer, STR_SIZE+10);
16. }

```

Fig. 3 Practice example program.

$PATH_v$ for any given input valuation n [21], [23]. Note that the $PATH_v$, and the `strncpy()` function may depend on the input valuation n . Definition 1 is used to determine the verification operation in program P . Based on this definition, we assume that the program control path includes a verification operation.

Definition 1 (Verification operation on the program control path)

Let V represent a set of input variables or parameters. For any given input variable v , program P includes a verification operation on the program control path. Let us assume a standard syntax for C source code statements, and define two statements, `stmt` and `verify_stmt`, to be equivalent if the syntax structures of the two statements are the same. That is, a `verify_stmt` is a semantic equivalence on a projection of `stmt` (denoted $stmt \approx verify_stmt$). Thus, the program control path, $PATH_v$, commences by starting with the verification operation, `verify_stmt()`, which occurs along the program control path based on the statement [27].

The value of any variable attainable by substituting the values of the input variables along the same control path can be verified by using Definition 2.

Definition 2 (Input variable valuation)

For every input variable v , let $V0$ signify the set of benign and valid variables corresponding to input variable v . $V0$ occurs along the control path $PATH_v$; i.e., $V0$ is the set of statements of the form $n := ne$ or $s := se$ executed by P on input valuation n . Let us assume that v and $V0$ are extracted on the same control path (i.e., $PATH_v = PATH_{V0}$) and we have the benign function $Fval: v \rightarrow V0, \{(v, Fval(v)) | v \in DOM^\dagger(Fval)\}$, and $Fval(v) = V0$. Thus, if there is a difference between the input variable v for P and $V0$, the input variable v is invalid [4].

It is possible to execute the verification operation for

any given input variable v to induce the control path. Note that if another statement $V0$ was known to us that could be executed on the same control path as v , this would enable VDL to execute the verification operation on v and $V0$ and then verify whether the statements contain vulnerabilities. We can therefore define benign function $Fval(v)$ as the set of statements $V0$ associated with every number and string variable. Note that if v and $V0$ induce the same control path, then $Fval(v) = V0$, which is a benign and safe variable corresponding to v for any statement. In addition, for any program P and input variable v , the value of any variable can be determined by using the values of the input variable in the statement for this variable. This mirrors the operation on the counterpart. It would be possible to analyze the semantic difference between v and $V0$ if $V0$ was to execute the same operation as v .

Example 3: Consider the program fragments in Fig. 3:

Given the input v : ($n = 1$, `org_buffer` = “Peter Pan” and `dest_buffer` = “Harry Potter”), the program executes and then generates a statement. Consider the benign function $Fval()$ and safe representation variable $V0$:

- Vulnerable: `strncpy(org_buffer, dest_buffer, 20);`
 $Fval(org_buffer) = REP_org_buffer$,
 $REP_org_buffer.val = \text{“Peter Pan,”}$
 $REP_org_buffer.len = 9$,
 $REP_org_buffer.maxlen = 10$.
 $Fval(dest_buffer) = REP_dest_buffer$,
 $REP_dest_buffer.val = \text{“Harry Potter,”}$
 $REP_dest_buffer.len = 12$,
 $REP_dest_buffer.maxlen = 20$.

Transforming $V0$ into the statement yields:

- Safe: `verify_stmt(strncpy, REP_org_buffer, REP_dest_buffer, 20);`

Continuing from Definition 2, we can also define a variable property that is benign and safe and that corresponds to v for any statement to verify the buffers as a pair of integer ranges.

Definition 3 (Variable constraints and assertions)

We model the operation of each string s in terms of its effect on two buffer attributes: the number of bytes declared for the buffer ($s.maxlen$) and the number of bytes currently in use ($s.len$) [28]. Therefore, the safe property of $s.maxlen > s.len$ for all variables s should be verified.

In terms of variables, a latent vulnerability is safe if all the constraints and assertions are TRUE [29], when these variables are subjected to a sanitization process. For example, a `strncpy(dest, src, num)` function with a constraint ($\min(src.len, num) \subseteq dest.len$) and an assertion ($dest.maxlen \geq \min(src.len, num)$) is generated. The constraint means that `dest.len` should be replaced with $\min(src.len, num)$ and the assertion means that if $dest.maxlen < \min(src.len, num)$

[†]We use DOM as the DOMAIN for the function $Fval()$.

Table 2 Examples of string operations, constraints, and assertions of variables.

Variable description	C functions	Constraints	Assertions
Pointer variables	char *p	$0 \subseteq p.len$	$p.maxlen \geq 0; p.len \geq 0$
	p = malloc(n)	$n \subseteq p.len$	$p.maxlen \geq n; p.len \geq 0$
	p = *gethostbyname(name)	$0 \subseteq p \rightarrow h_name.len; name \subseteq alloc(p)$	$p.len \geq 0$
	p = strchr(src, char)	$char.len \subseteq src.len; src \subseteq alloc(p)$	$src.maxlen \geq char.len; p.len \geq 0$
	p = strrchr(src, char)	$char.len \subseteq src.len; src \subseteq alloc(p)$	$src.maxlen \geq char.len; p.len \geq 0$
	p = strstr(src, str)	$str.len \subseteq src.len; src \subseteq alloc(p)$	$src.maxlen \geq str.len; p.len \geq 0$
	p = memchr(src, char, n)	$char.len \subseteq src.len + n; src \subseteq alloc(p)$	$src.maxlen \geq \min(char.len, n); p.len \geq 0$
Representative variables	REP_CHAR a[n]	$n \subseteq a.maxlen; 0 \subseteq a.len$	$a.maxlen \geq n$
	REP_CHAR a[n] = x	$\min(a.len, n+1) \subseteq a.len$	$a.maxlen \geq \max(x.len, n)$
Buffer index variables	strcpy(dst, src)	$src.len \subseteq dst.len$	$dst.maxlen \geq src.len$
	strncpy(dst, src, n)	$\min(src.len, n) \subseteq dst.len$	$dst.maxlen \geq \min(src.len, n)$
	strcat(dst, src)	$src.len + dst.len \subseteq dst.len$	$dst.maxlen \geq dst.len + src.len$
	strncat(dst, src, n)	$dst.len + n \subseteq dst.len$	$dst.maxlen \geq dst.len + n$
	sprintf(dst, "%s", src)	$src.len \subseteq dst.len$	$dst.maxlen \geq src.len$
	sprintf(dst, "%d", n)	$n \subseteq dst.len$	$dst.maxlen \geq n$
	snprintf(dst, n, "%s", src)	$\min(src.len, n) \subseteq dst.len$	$dst.maxlen \geq \min(src.len, n)$
	vsprintf(dst, "%s", arg_list)	$arg_list.len \subseteq dst.len$	$dst.maxlen \geq arg_list.len$
	strncpy(dst, src)	$src.len \subseteq dst.len$	$dst.maxlen \geq src.len$
	dst = strdup(src)	$src.len \subseteq dst.len$	$dst.maxlen \geq src.len$
	strxfrm(dst, src, n)	$\min(src.len, n) \subseteq dst.len$	$dst.maxlen \geq \min(src.len, n)$
	memcpy(dst, src, n)	$\min(src.len, n) \subseteq dst.len$	$dst.maxlen \geq \min(src.len, n)$
	memmove(dst, src, n)	$\min(src.len, n) \subseteq dst.len; n \subseteq dst.len$	$dst.maxlen \geq \max(src.len, n)$
	r = strcmp(dst, src)	$src.len \subseteq dst.len$	$dst.maxlen \geq src.len; -1 \leq r \leq 1$
	r = strncmp(dst, src, n)	$\min(src.len, n) \subseteq dst.len$	$dst.maxlen \geq \min(src.len, n); -1 \leq r \leq 1$
	memset(dst, const, n)	$const.len * n \subseteq dst.len$	$dst.maxlen \geq \max(const.len * n, n)$
	itoa(src, dst, radix)	$radix(src.val).len \subseteq dst.len$	$dst.maxlen \geq src.val.len$
	strlen(str)	$0 \subseteq str.len$	$str.maxlen \geq str.len$
	scanf("%ns", src)	$n \subseteq src.len$	$src.maxlen \geq n$
	sscanf(dst, "%s", arg_list)	$arg_list.len \subseteq dst.len$	$dst.maxlen \geq arg_list.len$
	gets(n src)	$n \subseteq src.len$	$src.maxlen \geq n$
	puts(n src)	$n \subseteq src.len$	$src.len \geq n$
	fgets(dst, n, src)	$\min(src.len, n) \subseteq dst.len$	$dst.maxlen \geq \min(src.len, n)$
	fputs(dst, src)	$src.len \subseteq dst.len$	$dst.maxlen \geq src.len$
	dst = getenv(arg_list)	$arg_list.len \subseteq dst.len$	$dst.maxlen \geq arg_list.len$

is *TRUE* then this is a fault.

Table 2 presents an example of some typical string operations and constraints/assertions. The second column lists the C functions in which we are interested, whereas the constraints and assertions are included in the third and fourth columns, respectively.

5. Implementation

5.1 The Tool: VDL

Figure 4 presents an overview of the tool with which we implemented the candidate code generation technique. Our tool consists of the following two components: an offline C program transformer and a semantic substitution code estimator. The former is implemented using the C source

Fig. 4 Overview of VDL (a) offline layer (b) run-time layer.

transformation tool. The latter is implemented using Behavior Knowledge (BK) analysis.

5.1.1 Transformation of the Source Code

The natural approach to implementing VDL is to transform the source code. This is accomplished in our prototype based on exact subtree matching, for which we use the Code Transformation Tool (CTT) [29], a transformation tool for source codes written in C. The CTT consists of two important phases: code selection and transformation phases. The code selection phase is carried out in a fingerprint database [30], [31], in which the structure is strictly specified to correspond to a specific pattern. Each fingerprint set (a set of subtrees represented with a digest form) that matches this pattern can latently be transformed. In the code transformation phase, the fingerprint sets that correspond to the specified structure and additional conditions are added to the new code. The substitution semantics of this code are the same as those of the original.

We designed VDL to search the fingerprint database to retrieve clusters (a group of each node corresponding to a statement) of exact subtree matches. To avoid F/P, we combined VDL with the method in [32]. As an example, we examined a specific number of child fingerprints, which VDL explores in a breath-first manner to reduce the probability of collision when discarding F/Ps in clusters [30]. For example, we considered two subtrees, rooted at nodes α and β with children $a_1, \dots, a_i$ and $b_1, \dots, b_i$, respectively. A comparison of the weight and hash values of the children pairs $(a_1, b_1), \dots, (a_i, b_i)$ becomes possible if α and β share weight and hash values [31]. We accomplished the retrieval through the parent node pointer of the fingerprint.

Iteration over the fingerprint database enables VDL to retrieve clusters that exactly correspond to the subtree. That

```

1. int main(void) {
2.     #define STR_SIZE 15
3.     FILE *pFile;
4.     char pBuffer[STR_SIZE+15];
5.     char cmp_buffer[STR_SIZE], max_buffer[STR_SIZE];
6.     REP_CHAR REP_pBuffer[STR_SIZE+15];
7.     REP_CHAR REP_cmp_buffer[STR_SIZE];
8.     REP_CHAR REP_max_buffer[STR_SIZE];
9.     ...
10.    while (fgets(pBuffer, STR_SIZE+15, pFile) != NULL) {
11.        verify_stmt(stncpy, REP_cmp_buffer, REP_pBuffer,
12.                    STR_SIZE+15);
13.        strncpy(cmp_buffer, pBuffer, STR_SIZE+15);
14.        if (strlen(cmp_buffer) > strlen(max_buffer)) {
15.            verify_stmt(stncpy, REP_max_buffer,
16.                        REP_cmp_buffer, STR_SIZE);
17.            strncpy(max_buffer, cmp_buffer, STR_SIZE);
18.        }
19.    }
20.    return 0;
21. }

```

Fig. 5 Transformed source code for the example in Fig. 1.

```

typedef enum FunctionTypes {
    None = 0,
    malloc = 1,
    strchr = 2,
    strrchr = 3,
    ...
    scanf = 53,
    gets = 54,
    puts = 55,
    fgets = 56,
    fputs = 57,
    getenv = 58
}

```

Fig. 6 Example of authorized functions.

is, the set S stored in the database presents us with different views on the code base functionality. In particular, we established the following two definitions for the set S [5]:

1. **API nodes:** The API node considers only the flat function and type names in the set S . The result set R simply contains all individual API nodes originating from the set S ; all other nodes are ignored.
2. **Subtrees:** The set R contains all subtrees of depth D that include at least a single API node.

Figure 5 shows the transformed code example for the program presented in Fig. 1. We fixed the depth of subtrees to $D = 3$ by using a default setting that is effective between a half-learned representation and complex subtrees [9]. This option remains useful for checking the shallow functionality of the source code.

5.1.2 Substitution Code Estimator Using Behavior Knowledge

It is worthwhile to state that all candidate variables should

Table 3 Simple if-else programming example of internal database rulesets for behavior information in BK according to string verification operation.

Rule description	Authorized function	Verification operation	Behavior information in BK
Example rule #1: Add a check to ensure that sufficient space is available to copy an object.	strcpy(org_buffer, dest_buffer);	verify_stmt(strcpy, REP_org_buffer, REP_dest_buffer);	/* +1 is for adding the null terminator */ if (REP_org_buffer.maxlen >= (REP_dest_buffer.len + 1)) strcpy(REP_org_buffer.val, REP_dest_buffer.val); else /* Buffer overflow vulnerability */ print("Buffer overflow %s", REP_dest_buffer.val);
Example rule #2: Add a check to ensure that destination buffer size has sufficient space.	strncpy(org_buffer, dest_buffer, n);	verify_stmt(strncpy, REP_org_buffer, REP_dest_buffer, n);	/* +1 is for adding the null terminator */ if (REP_org_buffer.maxlen >= REP_dest_buffer.len) strncpy(REP_org_buffer.val, REP_dest_buffer.val, n); else if ((REP_org_buffer.maxlen < REP_dest_buffer.len) && \ (REP_org_buffer.maxlen >= n + 1)) strncpy(REP_org_buffer.val, REP_dest_buffer.val, n); else /* Buffer overflow vulnerability */ print("Buffer overflow %s", REP_dest_buffer.val);
Example rule #3: Check buffer length before operation.	strcat(org_buffer, dest_buffer);	verify_stmt(strcat, REP_org_buffer, REP_dest_buffer);	/* +1 is for adding the null terminator */ if (REP_org_buffer.maxlen >= \ (REP_org_buffer.len + REP_dest_buffer.len + 1)) strcat(REP_org_buffer.val, REP_dest_buffer.val); else /* Buffer overflow vulnerability */ print("Buffer overflow %s", REP_org_buffer.val);
Example rule #4: Add a check to ensure that sufficient space is available to copy an object.	strncat(org_buffer, dest_buffer, n);	verify_stmt(strncat, REP_org_buffer, REP_dest_buffer, n);	/* +1 is for adding the null terminator */ if (REP_org_buffer.maxlen >= (REP_org_buffer.len + n + 1)) strncat(REP_org_buffer.val, REP_dest_buffer.val, n); else if ((REP_org_buffer.maxlen < (REP_org_buffer.len + n + 1)) \ && (REP_org_buffer.maxlen >= (REP_org_buffer.len \ + REP_dest_buffer.len + 1))) strncat(REP_org_buffer.val, REP_dest_buffer.val, n); else /* Buffer overflow vulnerability */ print("Buffer overflow %s", REP_org_buffer.val);
Example rule #5: Replace the character data type with string data type that does not overflow and can expand its size.	sprintf(org_buffer, "%s", dest_buffer);	verify_stmt(sprintf, REP_org_buffer, REP_dest_buffer, s);	/* +1 is for adding the null terminator */ if (REP_org_buffer.maxlen >= (REP_dest_buffer.len + 1)) sprintf(REP_org_buffer.val, "%s", REP_dest_buffer.val); else /* Buffer overflow vulnerability */ print("Buffer overflow %s", REP_org_buffer.val);

be included in the sanitization process for variables [4]. This idea of including a candidate variable in the source code differs from the traditional understanding of dynamic analysis, which is conducted after the execution of the source code. In our work, before the execution of the statement in the source code, VDL verifies the statement using the candidate variable. Thus, VDL can pre-examine the statement using a substitution variable before the real statement is executed.

We designed VDL to ensure the quality of the candidate variable and verified the statement by implementing the BK library, which represents the behavior information of the API statement in the control path of a program. The BK library is used to represent and store information about the analysis of statements in the source code of a program. Analysis of statements can also be represented by templates or descriptions used to analyze the program [4]. Behavioral patterns in the BK are connected by communication among the verification operations through an internal ruleset for detecting vulnerabilities. Before a statement is executed, BK compares the candidate variables for attack detection.

The *verify_stmt()*, i.e., the verification operation, calls the BK that compares the real variables with the candidate variables and verifies variable constraints and assertions (see Table 2). This procedure throws an exception if the candidate variables are not isomorphic. Otherwise, the original statement is issued during runtime. BK constraint conditions are generated by constantly dividing strings into sub-strings and then manipulating and combining them again. Therefore, we maintained a record of grammatical elements at the character rather than the string level. Next, we performed a syntax-aware evaluation of each code string before executing them.

Figure 6 shows an example of the authorized function text value that we are interested. We designed BK to use the following enumerator to identify authorized functions. Table 3 presents a simple *if-else programming* example of rulesets for string behavior information in BK according to the string verification operation. The second column lists the authorized function. The third column lists the verification operation according to the authorized function. The fourth


```

Get_Candidate_Costs (BK_patterns: behavior knowledge patterns;
                     REP_V1, ..., REP_Vn: substitute variables;
                     |N| : number of substitute variables)
1. Initial m = 0, i = 1, k = 1, p = 1
2. Read Verification_operation_stmt
3. m = |N| {current number of substitute variables}
4. For i = 1 to m
5.   Read_variables[i] = Read_params[i]
6.   Varseti[] = GenerateSubstitute (Read_variables[i])
7. Next i
8. For i = 1 to i(max)
9.   If (ComputeVarset (Varseti[]) <> NULL) Then
10.    RSstmt [k]=ComputeVarseti[] {variable result sizes}
11.    Increment k
12.   End if
13. Next i
14. CompareResultSizes (RSstmt [p])
End Procedure

```

Fig. 7 Brief scheme of the VDL-based estimation.

column presents the behavior information in BK according to the verification operation. Therefore, we can easily add or modify authorized function and behavior information in BK.

BK has two major steps: Step 1 analyzes three attributes of the grammatical structures of the codes and then determines all those with attributes relevant to the code instrumentation. There are three types of attributes: length, operation, and restraint. Step 2 analyzes irrelevant assertions to classify the vulnerability.

Figure 7 shows a brief scheme for the verification estimation. The BK calls the *Get_Candidate_Costs()* procedure, which compares the result between the real variables and those that are candidates. This procedure throws an exception if there is no equality in the results between the real and candidate variables.

Depending on the result obtained for the candidate variables, the estimation process is classified into three steps, in accordance with the phases in the algorithm:

- (1) Step1: Read the statements for the verification operation;
- (2) Step2: Analyze the preliminary stage in VDL algorithm;
- (3) Step3: Compute the resulting sizes for the real and candidate variables.

Step 1 involves reading the verification operation statements in the source code. Step 2 consists of inspecting each given predicate and analyzing the intended substitute statements in the operation statements. The *GenerateSubstitute()* function generates substitute statements and predicts the faults of the real variable. Step 3 computes the respective intermediate results before estimating the variable result. The variable result, which is calculated by using the *ComputeVarset()* function, is stored in *RSstmt[]*, after which

the respective *RSstmts* are compared with the aid of the *CompareResultSizes()* function.

5.2 Limitations and Extensions

Despite the effectiveness of our technique in detecting vulnerabilities, many situations exist in which failure can occur. First, some situations require developer intervention. For example, ambiguity may be detected in a source code. In this case, we recommend rewriting the violating statements.

Second, our technique does not consider user-defined API libraries. Therefore, analysis of programs with user-defined libraries would result in false negatives.

Third, our aim is to detect code vulnerabilities automatically and to prevent vulnerabilities by analyzing the verification operation and customized database in an application. Therefore, we concentrated on ensuring that the detection of vulnerabilities in the source code is accurate.

Fourth, our approach can also be used to process user-defined functions with substitution variables. These represent sets of operations stored within the actual program. For example, assuming that *char *ants (char *pr)* is a prototype of a user-defined function, we additionally use the variables *fun_ants.val*, *fun_ants.len*, and *fun_ants.maxlen* as the return values for the user-defined function *ants*, and also add *ants-pr.val*, *ants-pr.len*, and *ants-pr.maxlen* as attributes of parameter *pr*. All of these are added as global variables. This requires the parameter attributes to be assigned at the moment when the function is called, and to assign the attributes of the return value when the function is returned. That is, the status before the call should be recorded, and the callee function should be maintained and then used for analysis.

6. Evaluation

We defined two dimensions to evaluate our approach: (a) conducting empirical experimentation and (b) maintaining performance overheads at an acceptable level.

6.1 Empirical Evaluation of the Test-Suite

We evaluated the VDL verification by using the Juliet test set [33], which is a software assurance reference dataset, to perform empirical experiments with application datasets. Several other techniques have also been evaluated using this test set [34]. The test set is a collection of C/C++ programs with known faults consisting of 32,099 test cases and covering 90 different kinds of faults documented in the Common Weakness Enumeration (CWE) [9]. Simple code examples with security vulnerabilities are provided as well as instances in which the fault is embedded in various control flow and data-flow patterns. The test set consists of test cases of two types: the illegit set, which comprises faults, and the legit set, which has legitimate inputs that have the appearance of faults. We tested the illegit set to

Table 4 Results of the vulnerability evaluation using the Juliet test suite.

Security model	CWE No.	Size (LOC)	AVO		Test-cases		Success/ flaws	Parse errors	F/P	Vulnerability	
			API nodes	Subtrees	Illegit set	Legit set				API nodes	Subtrees
Buffer handling	CWE-190	3,214	119	151	110	21	86/86	1	0	0	1
	CWE-191	2,521	113	148	90	15	46/46	0	0	13	0
	CWE-120	2,190	123	149	151	21	89/89	21	3	1	11
	CWE-121	2,232	109	139	119	48	101/101	0	0	0	0
	CWE-131	1,832	121	158	154	11	121/121	22	2	15	9
Input validation	CWE-129	8,332	307	289	391	182	281/281	27	2	15	15

check whether our VDL is capable of successfully preventing buffer overflow vulnerabilities. In addition, we tested the legit set to ascertain whether any F/Ps are discovered. (F/Ps are obtained when the VDL modifies inputs from the legit set.)

The CWE entries were combined in more abstract categories[†]. For example, a category such as “Buffer overflow” represents different CWE entries that describe the type of buffer overflow (e.g., CWE-121 Stack-based buffer overflow, CWE-190 Integer overflow, etc). Therefore, we used security model categories in accordance with the “Center for Assured Software 2011” to generate a security model as part of a more general software quality model [35]. This security model facilitates interpretation of the Juliet test set results and thereby provides a connection between generic descriptions of software security attributes and specific software analysis approaches.

The Juliet code flaw test set was executed to verify whether the VDL can successfully detect faults. The Juliet code flaw test set was run to check whether the loss of precision causes F/P or false negatives. The illegit suite contains more than 50 different illegit cases, such as buffer overflow and memory leak. These inputs include data that can destroy the input validation techniques of an application. Table 4 summarizes the results of the vulnerability evaluation using the Juliet test suite. The category “Buffer handling” has the most test cases because problems in this area are common security issues in C/C++ programs. The third column lists the Line of Code (LOC). The fourth column refers to the number of Analyzed Verification Operations (AVO) in the security model that issues our *verify_stmt()* operations to the source code. The “Test-cases” column shows the number of test cases each set contains. The sixth column lists the number of successful flaw attempts on the original test sets and that of flaws detected by the VDL. Note that none of the unsuccessful flaws and none of the legitimate inputs were modified by the VDL. The “Parse errors” column shows the number of parse errors that necessitate human intervention to perform selected control flows. The eighth column shows the number of F/Ps because there are situations in which our VDL fails due to parse errors (see Sect. 5.2.1). The final two columns show the number of vulnerabilities in the security

model that issues our *verify_stmt()* operations to the source code.

6.2 Evaluation of Real-World Test Suite

To evaluate the VDL verification, we used Common Vulnerabilities and Exposures (CVE) when describing vulnerabilities to reproduce the reported exploits^{††}. CVE provides a reference method for publicly known information on security vulnerabilities and exposures, as reported by software development organization, individuals and coordination centers [36]. We performed experiments to validate whether the VDLs successfully detect flaws and vulnerabilities.

To effectively study the information associated with buffer overflow vulnerabilities, we ran the five real world open-source projects in CVE. The following describes the code bases of these projects: Pidgin, Rsync, LibTiff, FFmpeg, and Idera-uptime. The following describes the code bases of these projects:

- (1) Pidgin is an instant messaging client that implements a range of communication protocols. Version 2.11.0 of the client contains MXIT multiple buffer overflow vulnerabilities. Consequently, data is copied without verifying that it was copied successfully.
- (2) Rsync provides file transfer capabilities. Rsync 3.x versions before 3.0.8 contain a vulnerability that enables remote Rsync servers to exercise denial of service operations (heap memory corruption and application crash). Alternatively, these servers could possibly execute arbitrary code by way of corrupted data.
- (3) LibTiff is a reading and writing Tagged Image File Format (TIFF) files library. Version 3.8.1 of the library consists of a buffer overflow (stack based) for parsing MSVR elements.
- (4) FFmpeg produces multimedia data libraries and programs. Version 2.8.0 has been found to contain a vulnerability that enables arbitrary code execution because of buffer overflow in *decode_dds1* function.
- (5) Idera-uptime is an infrastructure monitor. Version 7.04 has been found to contain buffer overflow errors. Con-

[†]The MITRE Corporation, 2018, <http://cwe.mitre.org/>.

^{††}The MITRE Corporation, 2018, <http://cve.mitre.org/>.

Table 5 Results of the evaluation result using the real-world test-suite.

Application	CVE No.	Size (LOC)	AVO		Parse errors	F/P	Vulnerability		VDL detection
			API nodes	Subtrees			API nodes	Subtrees	
Pidgin	CVE-2016-2368	272,866	918	1,215	5	3	5	8	Success
Rsync	CVE-2015-0932	52,961	208	237	2	0	1	3	Success
LibTiff	CVE-2016-9536	52,650	181	198	1	1	2	5	Success
FFmpeg	CVE-2016-10191	41,723	119	163	4	2	0	3	Success
Idera-uptime	CVE-2015-2895	85,121	519	392	5	3	3	1	Success

```

1. if ((int) strlen(offSideResult) > 0) {
2.   ...
3.   strcpy(onSideResult, offSideResult);
4.   ...
5. }

```

(a) Example of buffer overflow fault in Idera-uptime application.

```

1. if ((int) strlen(offSideResult) > 0) {
2.   ...
3.   verify_stmt strcpy, REP_onSideResult,
4.   REP_offSideResult);
5.   strcpy(onSideResult, offSideResult);
6.   ...
7. }

```

(b) Example of safe Idera-uptime source code transformed by VDL.

Fig. 8 Detecting buffer overflow fault in Idera-uptime application.

sequently, the Idera-uptime client allows remote attackers to execute arbitrary code via long command input.

Table 5 summarizes the results of evaluation using the open-source test suite. The second column refers to CVE identifiers of vulnerabilities present in these applications. The application size is given in the LOC. AVO lists the verification operation in the application that issues *verify_stmt()* operations to the source code. The “Parse errors” column shows the number of parse errors that require human intervention to transform the control flow and source code. The sixth column shows the number of false positives that result in a situation in which our VDL fails (see Sect. 5.2.1). In addition, the false positive rate was measured by manually analyzing known CVE vulnerabilities for each application. The final two columns list the number of vulnerabilities detected by our VDL sanitizer.

Figure 8(a) presents an example of a discovered buffer overflow vulnerability in the Idera-uptime program. The *strcpy()* function copies the input (*offSideResult*) to a locally declared array identified by *onSideResult* without checking the destination buffer size. Therefore, this statement results in buffer overflow faults for *onSideResult*. Figure 8(b) presents an example of a safe source code transformed by VDL using a verification operation.

Figure 9(a) presents an example of an integer overflow fault in the LibTiff application. This source code assumes that the *img → sizey* is passed from the argument

```

1. get_MSVR_element(pp, &width, &height, ...) {
2.   ...
3.   /* struct containing img data, 10KB each */
4.   img_t table_ptr;
5.   ...
6.   img→scalex=width;
7.   img→sizey=height;
8.   table_ptr=malloc(img→scalex * img→sizey * 3);
9.   ...
10. }

```

(a) Example of integer overflow fault in LibTiff application.

```

1. get_MSVR_element(pp, &width, &height, ...) {
2.   ...
3.   /* struct containing img data, 10KB each */
4.   img_t table_ptr;
5.   ...
6.   img→scalex=width;
7.   img→sizey=height;
8.   verify_stmt(malloc, table_ptr,
9.   REP_img→scalex * REP_img→sizey * 3);
10.  table_ptr=malloc(img→scalex * img→sizey * 3);
11.  ...
12. }

```

(b) Example of safe LibTiff source code transformed by VDL.

Fig. 9 Detecting integer overflow fault in LibTiff application.

&height. If *img → sizey* is given a large enough value, the operation “*img → sizey*3*” may overflow first. For example, let us assume that *img → sizey* = 0x60000000, the “*img → sizey*3*” will be equal to 0x20000000 (overflow). Figure 9(b) presents the safe source code transformed by VDL.

6.3 Accuracy and Performance

Our proposed technique was compared with four well-known existing detection approaches: Cppcheck (Version 1.83), Flawfinder (Version 2.0.6), Valgrind (Version 3.13) and Splint (Version 3.1.2) [37]. The extent to which faults are accurately detected or bugs prevented is represented by the number of vulnerabilities that are accurately detected. We implemented the technique by computing the detection accuracy and the overhead imposed by the approaches. The server was an Ubuntu 16.04 Linux machine with a 2.40 GHz Intel®4-Cores™ i7 processor and 32.00 GB RAM. We obtained the sum of the user and system time by using the *time*

Fig. 10 Comparison of detection accuracy obtained for various algorithms.

Table 6 Comparison of algorithms with respect to deployment requirements.

Algorithm	Detection	Prevention	Time Complexity	Usability
Cppcheck	Yes	Yes	Medium	Medium
Flawfinder	Yes	No	Low	Low
Valgrind	Yes	Yes	High	High
Splint	Yes	Yes	High	High
VDL	Yes	Yes	Very high	High

command. Network delay was avoided by installing the applications on the local host.

Figure 10 shows the detection accuracy for the six sample datasets (Juliet and real-world test suite). We performed sample runs and measured the average response time for each run in the case of each test set by disabling caching on the servers. We ensured accuracy by running our timing experiments three times and utilized the average runtime. The Cppcheck tool can detect fixed-sized local buffer and bounds checking, but it does not detect incorrect buffer size calculations. The Flawfinder tool only considers faults related to a fixed-sized local buffer. That is, it only detects fixed-size buffer overflow statements. The Valgrind tool can detect buffer errors, invalid conversions of error messages, and out of bounds errors, but does not detect context errors in memory leaks. Like Valgrind, Splint can detect most types of buffer overflow errors, but cannot detect improper operations within the bounds of a buffer. In contrast, the VDL tool is effective in preventing and detecting buffer overflow flaws. Moreover, because the VDL algorithm incorporates several of the model construction checking steps, it can prevent and detect faults both exactly and accurately. Therefore, the accuracy of the VDL algorithm was 10–23% higher compared with that of the Cppcheck, Flawfinder, Valgrind, and Splint tools.

In addition to detection time, time complexity and effort are other important factors for evaluating a detection system. Table 6 compares the algorithms with respect to deployment requirements. The term ‘High’ has twice the time complexity of the term ‘Medium’. In addition, the term ‘Very high’ has twice the time complexity of the term ‘High’. It is evident that the VDL tool consumed more time for detection than existing methods. This is because the VDL tool includes a number of steps that consider semantic methods to produce more accurate results. Our results are of significance, however, in that we have developed a novel system capable of detecting C/C++ source code buffer overflow vulnerabilities more precisely than previous approaches. Additionally, the system is based on accurate mechanisms for detecting and preventing source code vulnerabilities, which are usually time-consuming tasks [37]. Therefore, we do not focus on the time complexity of the VDL tool in this paper. We have demonstrated that this is feasible, and our evaluation presents reasonable overheads in an experimental implementation.

7. Conclusion and Future Work

Most program errors stem from unexpected input values and the debugging of errors requires significant amounts of personal and physical resources. Flaws in software design and faults in program code are constantly being introduced into programs during development. This study led to our development of a novel approach to verify and sanitize code instrumentation by comparing the result for each candidate variable with that expected from the input data in C/C++ programs. The comparison is not only based on analysis of the information needed to check faults in the statements but is also a comparison of the intended statements using substitution code evaluation. Our results showed that programs containing buffer overflow vulnerabilities could be detected and prevented through the generation of substitution code and the sanitization of code vulnerabilities based on candidate variables. This ensures that faults can be detected prior to execution of the statement, thereby preventing malicious access. Our approach is particularly useful in enhancing software security monitoring, thus enabling us to successfully develop retrofitting techniques that guarantee secure applications and eliminate faults.

We implemented our approach and demonstrated its effectiveness in correcting buffer overflow flaws by evaluating the substitution code. The current study can be differentiated from previously published reports in the following two respects:

- (1) We attempted to verify the extent to which the sanitization process is correct and thereby identifies vulnerabilities using a substitution code generation technique.
- (2) Our system additionally performs a separate dynamic analysis to discover input values that are closely associated with buffer overflow vulnerabilities.

Our results are of significance in that we have

developed a novel technique to detect buffer overflow vulnerabilities by generating substitution code. The proposed technique is expected to contribute to the role of software security in software development. However, further studies are warranted to assess the efficiency of the technique through modification of the algorithm.

Acknowledgements

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (2015-0-00445) supervised by the IITP (Institute for Information & communications Technology Promotion) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00532, Development of High-Assurance ($\geq$ EAL6) Secure Microkernel).

References

- [1] J. Stark, "Product lifecycle management," *Product Lifecycle Management*, Springer, vol.1, pp.1–29, 2015.
- [2] C.-C. Huang, F.-Y. Lin, F.-Y.-S. Lin, and Y.S. Sun, "A novel approach to evaluate software vulnerability prioritization," *J. Syst. Softw.*, vol.86, no.11, pp.2822–2840, 2013.
- [3] F. Gao, L. Wang, and X. Li, "BovInspector: automatic inspection and repair of buffer overflow vulnerabilities," *Proc. 31st IEEE/ACM Int. Conf. Automated Software Engineering*, pp.786–791, 2016.
- [4] P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," *ACM Trans. Info. Syst. Security*, vol.13, no.2, pp.1–38, 2010.
- [5] B.V. Deokate and D.B. Hanchate, "Software source code plagiarism detection using latent semantic analysis," *Int. J. Sci Res.*, vol.5, pp.1059–1063, 2006.
- [6] H. Shahriar, H.M. Haddad, and I. Vaidya, "Buffer overflow patching for C and C++ programs: rule-based approach," *SIGAPP Appl. Comput. Rev.*, vol.13, no.2, pp.8–19, 2013.
- [7] Y. Yang, C. Onita, X. Zhang, and J. Dhaliwal, "TESTQUAL: Conceptualizing software testing as a service," *e-Service J.*, vol.7, no.2, pp.46–65, 2011.
- [8] B.M. Padmanabhuni and H.B.K. Tan, "Auditing buffer overflow vulnerabilities using hybrid static-dynamic analysis," *IET Software*, vol.10, no.2, pp.54–61, 2016.
- [9] A.E. Elhadi, M.A. Maarof, and B.I. Barry, "Improving the detection of malware behaviour using simplified data dependent API call graph," *Int. J. Sec. Appl.*, vol.7, no.5, pp.29–42, 2013.
- [10] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," *ICST IEEE Int. Conf.*, pp.91–101, 2016.
- [11] H. Brar and P. Kaur, "Comparing detection ratio of three static analysis tools," *Int. Journal of Computer Applications*, vol.124, no.13, pp.35–40, 2015.
- [12] Flawfinder - C/C++ Source Code Analyzer, <http://www.dwheeler.com/flawfinder/>, 2017.
- [13] E. Cilia, R. Pancsa, P. Tompa, T. Lenaerts, and W. Vranken, "From protein sequence to dynamics and disorder with DynaMine," *Nat. Commun.*, vol.4, pp.1–10, 2013.
- [14] P. Godefroid, M.Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol.10, no.1, pp.40–44, 2012.
- [15] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," *Proc. 22nd USENIX Conf. Security*, pp.49–64, 2013.
- [16] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," *Proc. 2015 10th Joint Meeting on Foundations of Software Engineering*, pp.214–224, 2015.
- [17] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. Progr. Lang. Syst.*, vol.27, no.3, pp.477–526, 2005.
- [18] A. Kiezun, V. Ganes, S. Artzi, P. Guo, P. Hooimeijer, and M. Ernst, "HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars," *ACM T. Softw. Eng. Meth.*, vol.21, no.4, pp.25–52, 2012.
- [19] P. Soulier, D. Li, and J.R. Williams, "A survey of language-based approaches to cyber-physical and embedded system development," *Tsinghua Science and Technology*, vol.20, no.2, pp.130–141, 2015.
- [20] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, pp.67–96, 2012.
- [21] G. Vidal, "Concolic execution and test case generation in Prolog," *Proc. 24th Int. Sym. on Logic-Based Program Synthesis and Transformation*, vol.8981, pp.167–181, 2015.
- [22] K. Jothisna and R.V. Krishniah, "A signature-free buffer overflow attack blocker using genetic programming," *Int. J. Emerg. Tech. Adv. Eng.*, vol.3, no.2, pp.640–647, 2013.
- [23] S.J. Salunkhe and U.L. Kulkarni, "Software theft detection for JavaScript programs based on dynamic birthmark extracted from runtime heap graph," *IOSR J. Comput. Eng.*, vol.17, no.3, pp.34–39, 2015.
- [24] D.A. Plaisted, "Source-to-source translation and software engineering," *J. Softw. Eng. Appl.*, vol.6, no.4, pp.30–40, 2013.
- [25] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM SIGPLAN Notices*, vol.47, no.6, pp.193–204, 2012.
- [26] Y.-S. Jang and J.-Y. Choi, "Detecting SQL injection attacks using query result size," *Comput. Sec.*, vol.44, pp.104–118, 2014.
- [27] E. Ferneley, "Coupling and control flow measures in practice," *J. Syst. Softw.*, vol.51, no.2, pp.99–109, 2000.
- [28] M. Valera, K. Rathod, and U. Chauhan, "A neoteric web recommender system based on approach of mining frequent sequential pattern from customized web log preprocessing," *Int. J. Comput. Appl.*, vol.69, no.24, pp.6–21, 2013.
- [29] M. Boekhold, I. Karkowski, and H. Corporaal, "Transforming and parallelizing ANSI C programs using pattern recognition," *Int. Conf. on High-Performance Computing and Networking*, pp.673–682, 2006.
- [30] D. Gugerli, "The world as database: on the relation of software development, query methods, and interpretative independence," *Info. Culture*, vol.47, no.3, pp.288–311, 2012.
- [31] S. Ransbotham and S. Mitra, "Choice and chance: a conceptual model of paths to information security compromise," *Infor. Syst. Res.*, vol.20, no.1, pp.121–139, 2009.
- [32] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," *2009 IEEE 17th Int. Conf. on Program Comprehension*, pp.243–247, 2009.
- [33] NIST, Juliet test suite for C/C++ v1.3, <https://samate.nist.gov/SRD/testsuite.php>, 2017.
- [34] A. Nanthamornphong and J.C. Carver, "Test-driven development in scientific software: a survey," *Softw. Quality J.*, vol.25, no.2, pp.343–372, 2017.
- [35] D. Port and J. Wilf, "The value proposition for assurance of JPL systems," *Procedia Comput. Sci.*, vol.28, pp.398–403, 2014.
- [36] L.A.B. Sanguino and R. Uetz, "Software vulnerability analysis using CPE and CVE," *arXiv preprint arXiv, 1705.05347*, 2017.
- [37] M. Belaoucha, D. Barthou, A. Eliche, and S.-A.-A. Touati, "FADAlib: an open source C++ library for fuzzy array dataflow analysis," *Procedia Comput. Sci.*, vol.1, no.1, pp.2075–2084, 2010.

Young-Su Jang received M.S. degree in Computer Science from Korea University, Seoul, Korea, in 2011. Currently, he is a Ph.D. candidate with the Computer Science and Engineering Department, Korea University, Seoul, Korea. His current research interests are secure software engineering, secure coding, and security.

Jin-Young Choi received M.S. degree from Drexel University, Philadelphia, PA, in 1986, and Ph.D. degree from the University of Pennsylvania, Philadelphia, in 1993. He is currently a Professor with the Computer Science and Engineering Department, Korea University, Seoul, Korea. His current research interests are real-time computing, formal methods, programming languages, process algebras, security, and software engineering.