Feng-Juan Gao, Yu Wang, Lin-Zhang Wang *et al.* Journal of computer science and technology: Instruction for authors. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 1–33 January 2018. DOI 10.1007/s11390-015-0000-0

Automatic Buffer Overflow Warning Validation

Feng-Juan Gao[1,2], Yu Wang[1,2], Lin-Zhang Wang[1,2*], *Distinguished Member, CCF*, Zijiang Yang[3], *Senior Member, IEEE*, and Xuan-Dong Li[1,2], *Fellow, CCF*

[1] *State Key Laboratory of Novel Software Technology, Nanjing University, Naning 210023, China*

[2] *Department of Computer Science and Technology, Nanjing University, Naning 210023, China*

[3] *Department of Computer Science, Western Michigan University, Kalamazoo, Michigan 49008-5466, USA*

E-mail: {fjgao,yuwang_cs}@smail.nju.edu.cn; lzwang@nju.edu.cn; zijiang.yang@wmich.edu; lxd@nju.edu.cn

Received July 15, 2018 [Month Day, Year]; revised October 14, 2018 [Month Day, Year].

Abstract Static buffer overflow detection techniques tend to report too many false positives fundamentally due to the lack of software execution information. It is very time consuming to manually inspect all the static warnings. In this paper, we propose BovInspector, a framework for automatically validating static buffer overflow warnings and providing suggestions for automatic repair of true buffer overflow warnings for C programs. Given the program source code and the static buffer overflow warnings, BovInspector first performs warning reachability analysis. Then, BovInspector executes the source code symbolically under the guidance of reachable warnings. Each reachable warning is validated and classified by checking whether all the path conditions and the buffer overflow constraints can be satisfied simultaneously. For each validated true warning, BovInspector provides suggestions to automatically repair it with 11 repair strategies. BovInspector is complementary to prior static buffer overflow discovery schemes. Experimental results on real open source programs show that BovInspector can automatically validate on average 60% of total warnings reported by static tools.

Keywords buffer overflow, static analysis warning, symbolic execution, automatic repair

1 Introduction

Buffer overflow occurs when more data is written into a buffer than the buffer capacity, causing extra data being written into memory adjacent to the buffer. If the adjacent memory before being overwritten has stored information (such as the pointer to the previous frame and return address) that is critical for the OS to correctly execute programs, buffer overflow may cause unpredictable behavior. In a buffer overflow attack, the attacker carefully crafts his/her input data to vulnerable software so that the unpredictable behavior is that the OS executes his malicious code embedded in the overflow data with the privilege of the vulnerable software.

Although more than forty years have passed since the buffer overflow technique was first documented by Anderson in 1972 [1] and almost thirty years have passed since the buffer overflow technique was first exploited by the infamous Morris worm in 1988, buffer overflow remains the most common type of software vulnerabilities, as shown in the recent studies of software vulnerability databases [2], and it is likely to remain so for many years to come. Most existing software has buffer overflow vulnerabilities, which are unknown

Regular Paper

Special Section on Software Systems 2020

A preliminary version of the paper was published in the Proceedings of ASE 2016.

This work was supported by the National Natural Science Foundation of China under Grant No.62032010, and partially supported by Postgraduate Research & Practice Innovation Program of Jiangsu Province of China.

*Corresponding Author

to their vendors and users, but could be exploited by attackers sooner or later. Most future software will still be written by programmers who are not well trained in software security. The inherently unsafe languages C and C++ will remain popular languages for performance and backward compatibility reasons. Although we have known how to avoid buffer overflow problems in writing programs for many years, having such knowledge is far from enough to thwart the rampant buffer overflow issue.

There are two general approaches to identifying buffer overflow vulnerabilities: static program analysis [3–12] and dynamic execution analysis [13–17]. The dynamic execution analysis approach needs a specific test case to trigger a buffer overflow vulnerability, which may not be easy to find. The static program analysis approach scans software source code to discover the code segments that are possibly vulnerable to buffer overflow attacks. Each vulnerability warning needs to be manually inspected to check whether each warning is indeed a true vulnerability. The key advantage of such schemes is that buffer overflow vulnerabilities can be discovered and fixed before software deployment. The key limitation of existing such schemes is that the reported buffer overflow vulnerabilities contain too many false positives fundamentally due to the lack of software execution information (such as which code segment is reachable in execution and which execution path will be followed) and each false positive wastes a huge amount of human effort on manual source code inspection.

From the above, people need methods to verify whether the warning reported by static program analysis approaches is true. Meanwhile, the test cases to trigger the buffer overflows are also required, which may help the programmers locate and fix the vulnerabilities.

In this paper, we propose BovInspector, a framework for automatically validating static buffer overflow warnings and providing suggestions for automatic repair of true buffer overflow warnings for C programs. Given the source code of a program, we first use static program analysis tools such as Fortify[1] to generate various software vulnerability warnings, including buffer overflow warnings. In this paper, our experiments were based on the buffer overflow warnings outputted by Fortify. For other static analyzers, BovInspector provides a unified static buffer overflow warning report format. It is straightforward to convert the output of another static analyzer to this format. For all the static buffer overflow warnings, BovInspector will classify them as true warnings, false warnings, and undecided warnings. True warnings are those where there exists at least one test case that triggers buffer overflow. For each true warning, BovInspector also outputs such a test case. In addition, BovInspector can provide suggestions to automatically repair the validated true warnings. False warnings are those where there exists no test case that triggers buffer overflow or that are simply unreachable. For such warnings, no human inspection is needed. Undecided warnings are those that BovInspector can neither find a test case to trigger buffer overflow nor prove that no such test case exists within a given time limit. The optimization goal of BovInspector is to minimize the number of undecided warnings. Ideally, we hope to minimize the number of undecided warnings to zero for all programs; however, this is theoretically impossible because the problem of using one program to find non-trivial properties of another program is fundamentally undecidable. Although BovInspector may report undecided warnings, it is still valuable because the validated safe paths and undecided paths with correspond-

[1] https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

ing path constraints and overflow constraints can help programmer to validate these warnings manually.

BovInspector is complementary to prior static buffer overflow discovery schemes. The key contribution of BovInspector is on eliminating the need to manually inspect the warnings that are actually false warnings, and providing automatic repair suggestions for validated true warnings, which will save a tremendous amount of manual efforts. In some sense, BovInspector is a practical post-processing procedure for static program analysis based buffer overflow vulnerability discovery schemes. Our experimental results on real open source programs show that BovInspector can automatically validate on average 60% of total warnings reported by the static tool Fortify. Typically, for each buffer overflow warning, there always exists a large number of possible paths from the entrance of the program to the buffer manipulation point that may trigger a buffer overflow, i.e., the warning point. Therefore, saving manual effort on inspecting one buffer overflow warning means saving manual effort on inspecting both the warning point and the plenty of paths associated with the warning point.

The key idea of BovInspector is to use symbolic execution to automatically identify those buffer overflow warnings that are true warnings or false warnings. The advantage of symbolic execution is its capability to explore program execution states that are unavailable to static program analysis. The disadvantage of symbolic execution is the infamous path explosion issue, i.e., the number of execution paths grows exponentially with the number of branching points. In BovInspector, to avoid path explosion, we use the warning paths to guide the symbolic execution so that we only focus on these warning paths. All execution paths that do not lead to warning paths are pruned.

In summary, this paper contributes the following:

- An automated framework for validating static buffer overflow warnings and providing repair suggestions for the true warnings.
- An open source tool for handling C programs[2].
- An evaluation to show the effectiveness and efficiency of our tool.

Compared to the demonstration paper [18], this work introduces formal definitions to provide guidelines for classifying buffer overflow warning paths and buffer overflow warnings. We then enhance the buffer overflow model by formalizing more possible APIs and classifying all the APIs into four types according to their functions. We also extend the method to provide 8 more kinds of repair strategies. Moreover, for each API in the buffer overflow model, we provide repair suggestions by applying all of its usable repair strategies in decreasing order of usage frequency. Meanwhile, we conduct more experiments on both synthetic and real-world programs to further demonstrate the effectiveness and efficiency of our method.

The rest of the paper proceeds as follows. We first introduce the background knowledge of symbolic execution and buffer overflow in Section 2. Then we present the formal description of static buffer overflow warning validation in Section 3. Then the overview of the approach is presented in Section 4. Next, we present technical details of the four modules of warning reachability analysis, guided symbolic execution, buffer overflow validation and targeted automatic repair suggestions in Sections 5, 6, 7, and 8, respectively. In Section 9, we show implementation details and experimental results of BovInspector. Then, we review related work in Section 10. Finally, we give concluding remarks in Section 11.

[2]BovInspector is available at `http://bovinspectortool1.github.io/project/`

2 Background

2.1 Symbolic Execution

Symbolic execution is a classical technique for software testing and analysis [19]. It is used to systematically test a program and generate test input with high coverage. Symbolic execution uses symbolic values as the input, instead of concrete input, to explore the execution space of a program. When symbolic execution encounters a branching condition, it forks the execution state, following both branch directions and updating the corresponding path constraints on the symbolic input. When it reaches a program exit or hits an error, the current path constraint will be solved to find a concrete test case that drives program execution to this program location. In this paper, we choose KLEE [20], a state-of-the-art open source tool as our symbolic execution engine.

The symbolic execution engine maintains an execution state pool and begins the execution with an initial execution state. The execution state contains information about that execution such as the memory model, program counters, symbolic variables, and path constraints. The symbolic execution engine fetches the program counter of the current execution from the execution state, and interprets the instruction pointed by the program counter. If the instruction refers to operations on symbolic variables, the symbolic variable information in the execution state is updated. If the instruction is a branch command, the symbolic execution engine will duplicate the current execution state, set the next instruction of the two execution states to the true direction and false direction of the branch instruction, and add corresponding condition constraints. The two execution states are then put back into the execution state pool. If the instruction refers to the program exit, the path constraints of corresponding execution state will be solved by a constraint solver to generate a concrete input that leads to the same path as the current execution travels. Then the execution state terminates and is removed from the execution state pool. Each time when the symbolic execution engine finished interpreting an instruction, it calls a searcher module to select an execution state from the execution state pool to decide the next execution to execute. The above procedure repeats until the execution state pool is empty.

2.2 Buffer Overflow

Buffer overflows, both on the stack and on the heap, are a major source of security vulnerabilities in C and C++ code. A buffer is a region of a physical memory storage used to temporarily store data. Using a buffer while programming in C and C++ makes a lot of sense and generally speeds up the calculation process, which may also lead to unexpected buffer overflow. A buffer overflow, or buffer overrun, occurs when more data is put into a fixed-length buffer than what the buffer can handle.

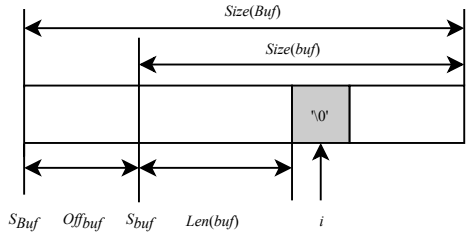

Fig.1. Buffer Model

There are two kinds of buffers we deal with, buffers of characters and buffers of the other values. We propose the buffer model shown in Fig.1. The symbols in the figure are defined as follows:

- S_{Buf}: The start address of buffer Buf.
- $Size(Buf)$: The available buffer space, in bytes, of buffer Buf starting from S_{Buf}.
- S_{buf}: The start address of buffer buf.
- Off_{buf}: The offset of bytes from the base address of the whole buffer of buffer buf, $Off_{buf} = S_{buf} - S_{Buf}$.

• $Size(buf)$: The available buffer space, in bytes, of buffer buf starting from S_{buf}.

• $Len(buf)$: The length of the content stored in buffer buf, which starts from S_{buf} and ends with terminator '\0'('\0'is excluded). To be noticed, if there is no terminator in buffer buf, we consider its length to be infinite in our model. $Len(buf)$ is only available for buffers of characters.

• i: i is the index of buffer buf when accessing $S_{buf}[i]$, which points to the ith element of buf. Each element in buf takes up a fixed number of bytes, marked as $typesize$.

For a buffer Buf with size $Size(Buf)$ and start address S_{Buf}, we assume a buffer operation accesses the data in the buffer buf and we know $buf = Buf + Off_{buf}$. Therefore, we can learn that the size of buf will be $Size(buf) = Size(Buf) - Off_{buf}$. The start address of buf will be $S_{buf} = S_{Buf} + Off_{buf}$. If the buffer operation is a direct buffer access, i.e., $S_{buf}[i]$, it will access the i-th element of buffer buf, namely the $(i + Off_{buf}/typesize)$-th element of buffer Buf at address $S_{Buf} + Off_{buf} + i \times typesize$. For a buffer of characters, a buffer operation may read the characters from it. The length of characters that will be accessed is $Len(buf)$.

3 Definition of Buffer Overflow Models and Warning Classifications

This section presents a lightweight formalism to define our buffer overflow model and the three categories of static buffer overflow warnings validated by our method.

We first define the simple imperative language (a core subset of C) in Fig.2 to represent all important features of C for the formal development.

Variables	$dest, src, buf \in \mathrm{V}$
Labels	$l \in \mathrm{L}$
Integers	$i, m, n, typesize \in \mathrm{I}$
Predicates	$\phi \in \Phi$
Statements	$e \in \mathrm{E}$

$$
\begin{aligned}
e ::= {}& {}^{l}buf = alloc(m * typesize) \mid {}^{l}buf[i] = x \mid {}^{l}e;\ {}^{l}e \mid \\
& {}^{l}bufAPI(dest, src, n, ...) \mid {}^{l}if(\phi)\ then\ {}^{l}e\ else\ {}^{l}e \mid \\
& {}^{l}buf = realloc(buf, m * typesize) \mid \\
& {}^{l}*(buf + i) = x
\end{aligned}
$$

Fig. 2. Imperative language for validating buffer overflow warnings

This language contains all important features of C that are necessary for us to validate a buffer overflow warning. Each statement e has a unique label $l \in \mathrm{L}$ that is used to identify e. In the following context, the statement label l for e means the line number of e. We use $buf = alloc(m * typesize)$ to represent an allocation of a new buffer with size $m * typesize$ in the stack or on the heap. For simplicity, we use $bufAPI(dest, src, n)$ to represent all kinds of buffer APIs, where $dest$ is a variable pointing to the destination buffer, src is a variable pointing to the source buffer and n, if provided by the API, indicates the number of bytes to be operated. If n is not supported by an API, then it means that n is unlimited. $bufAPI$ will copy the data with length n from src to $dest$. Whether these buffer operations will cause a buffer overflow depends on the remaining size of $dest$ buffer and the length of src content.

In a C/C++ program, a buffer is commonly used. There are also many buffer operations writing data into buffers in a program. As shown in the language, there are two categories of buffer operations: API call and direct buffer access. To characterize under what condition a buffer operation will lead to a buffer overflow, we propose our Buffer Overflow Models as follows.

Table 1. Buffer Overflow Models

Type	API	Parameter Format	Overflow Constraints
Unbounded content sensitive buffer operations	*strcpy*	(char* *dest*, const char* *src*)	$Len(src) \geq Size(dest)$
	strcat	(char* *dest*, const char* *src*)	$Len(src) + Len(dest) \geq Size(dest)$
	sprintf, *vsprintf*	(char* *str*, const char* *format*, ...)	$format\_string\_length \geq Size(str)$
	scanf, *vscanf*	(char* *format*, ...)	$format\_string\_length \geq Size(dest_i)$
	sscanf, *vsscanf*	(const char* *s*, char* *format*, ...)	$format\_string\_length \geq Size(dest_i)$
	fscanf, *vfscanf*	(FILE* *stream*, const char* *format*, ...)	$format\_string\_length \geq Size(dest_i)$
Bounded content sensitive buffer operations	*strncpy*, *snprintf* *vsnprintf*	(char* *dest*, const char* *src*, size_t *n*)	$n > Size(dest)$
	strncat	(char* *dest*, const char* *src*, size_t *n*)	$min\{Len(src), n\} + Len(dest) \geq Size(dest)$
	fgets	(char* *str*, int *num*, FILE* *stream*)	$num > Size(str)$
	fread	(void* *ptr*, size_t *size*, size_t *count*, FILE* *stream*)	$size * count > Size(ptr)$
	read	(int *fd*, void* *buf*, size_t *count*)	$count > Size(buf)$
Bounded content insensitive buffer operations	*memcpy* *memmove* *memset*	(char* *dest*, const char* *src*, size_t *n*)	$n > Size(dest)$
Direct buffer accesses	$buf[i]$ $*(buf + i)$	N/A	$(i+1) * typesize > Size(buf)$

Note: *format_string_length* is calculated by considering the impact of formatting symbols.

3.1 Buffer Overflow Models

According to the types of buffer operations, we divide buffer overflow models into two categories.

Buffer Overflow Models of API Call: For API calls, such as *strcpy* or *memset*, the buffer overflow can be detected by analyzing the parameters. We propose some constraint models for the APIs operating buffers in C99[3] and Linux system call interface[4]. To check for buffer overflow, we examine whether the data written to a buffer exceeds the buffer size. We list the constraints for different types of APIs shown in Table 1, in which $Len(src)$ and $Size(dest)$ are defined in Fig.1. Note that we select the parameter format of the first API in the group as a representative. Other APIs have a similar parameter format. We classify the APIs into four types, namely unbounded content sensitive buffer operations, bounded content sensitive buffer operations, bounded content insensitive buffer operations and direct buffer accesses.

Buffer Overflow Models of Direct Buffer Access: For direct buffer accesses in Table 1, we propose the following constraint models for array and pointer accesses. Accessing a buffer by array access can be represented as $buf[i] = x$, while *buf* represents a buffer, and x is the value to be assigned to the corresponding address. Accessing a buffer by pointer access can be represented as $*(buf + i) = x$. Without loss of generality, we also take type casting into consideration, e.g. when a pointer p points to a buffer `char` *buf*`[5]`, the buffer access $*((\text{int}*)p+1) = x$ will trigger a buffer overflow. Therefore, the buffer overflow condition is as follows:

$$(i+1) * typesize > Size(buf) \quad (1)$$

Based on the above buffer model and buffer overflow models, we introduce the definition of static buffer

[3] https://www.iso.org/standard/29237.html

[4] https://man7.org/linux/man-pages/man2/syscall.2.html

overflow warnings and the principles to classify these warnings, which will be used in our Buffer Overflow Validation module.

3.2 Buffer Overflow Warnings

Given the source code of a program, static analysis tools will locate all the statements that declare a buffer, track all the statements that perform operations on the buffers, and report a buffer overflow warning if the operation may violate the predefined secure coding rules. Before validating static buffer overflow warnings, we first give the definition of static buffer overflow warning as follows.

```
1  #define MAX_LEN  24
2  #define MIN_LEN  4
3  void usage() {
4      char des_buffer[MIN_LEN];
5      char* src_buffer="this is an example";
6      strcpy(des_buffer, src_buffer);
7  }
8  int initialize(char* argv_string) {
9      char mapped_argv[MIN_LEN];
10     if (strlen(argv_string) >= MAX_LEN)
11         return 0;
12     strcpy(mapped_argv,argv_string);
13     if (argv_string[strlen(argv_string)-1]
                                    != '-') {
14         strcat(mapped_argv, "-");
15     }
16     return MAX_LEN;
17 }

19 int main(int argc, char** argv) {
20     char* mode = (char*)malloc(MIN_LEN);
21     int len = 0;
22     if (strlen(argv[1]) < MIN_LEN)
23         len = initialize(argv[1]);
24     if (len > 0) {
25         mode = (char*)realloc(mode, len);
26     }
27     else {
28         mode[0] = argv[1][0];
29     }
30     mode[MIN_LEN-1] = '\0';
31     free(mode);
32     return 0;
33 }
```

Fig.3. An example program

Definition 1 (Static Buffer Overflow Warning). *A static buffer overflow warning ω is represented as a tuple: $(\langle l_1, l_2, \ldots, l_n\rangle, b)$, where l_1 is the label of the statement where a buffer is first declared or the entry of the main function, each $l_i\,(2 \leq i \leq n)$ is the label of the statement where an operation is performed on the buffer and b is the label of the statement where an overflow may occur on the buffer, which is regarded as a buffer warning point.*

Considering the source code of the example program in Fig.3, the static analysis tool reports four possible buffer overflow warnings that can be represented as ω_0=$(\langle 4\rangle, 6)$, ω_1=$(\langle 19, 23, 9\rangle, 12)$, ω_2=$(\langle 19, 23, 9, 12\rangle, 14)$, ω_3=$(\langle 25\rangle, 30)$.

For each buffer warning point b, there may be several paths from the entrance of the program to that point. We regard these paths as a warning path set.

Definition 2 (Warning Path Set). *For a static buffer overflow warning ω, a warning path set ps is a sequence $(P_1, P_2, \ldots, P_m)$, where each P_i is a set of path segments ρ, jointly constituting a path from the program entrance to the warning point b, and each path segment ρ records the statement label of the first statement of each basic block in the path. A ps represents all the complete paths from the entrance of the program to a buffer overflow point. The path constraint of P_i is $\bigwedge_{j=1}^{j=n_i} \varphi_j^i$, where n_i is the number of branch statements in path P_i, and φ_j^i is the constraint of the j-th branch in path P_i.*

Here we take as an example the warning ω_2=$(\langle 19, 23, 9, 12\rangle, 14)$ for the program in Fig.3. Its warning path set is $(\langle 19, 23, 8, 12, 14\rangle)$.

The key idea to validate a static buffer overflow warning ω is to find evidence that there exists a test case that follows one of the paths P_i in the warning path set ps, reaches the buffer warning point b, and finally triggers the buffer overflow. Therefore, at a buffer warning point b, BovInspector solves the conjunction of

path constraints and overflow constraints (denoted as function $OC(b)$), in order to validate the corresponding buffer overflow warning. $OC(b)$ is constructed and solved according to the *Overflow Constraints* in Table 1 regarding the API at the buffer warning point b. Based on the solving result, a buffer overflow warning will be validated as a true buffer overflow warning, a false buffer flow warning or an undecided buffer overflow warning.

For a static buffer overflow warning ω, a path P_i in the warning path set ps is an overflowable path if the conjunction of its path constraints and overflow constraints is satisfiable. If it is unsatisfiable, the path will be regarded as a safe path. If the solver cannot provide the result of the the constraints within the given time limit, the path will be regarded as an undecided path. For this case, we use $TIMEOUT$ to represent the situation that the constraint solver cannot provide the result of the constraints within the given time limit for path P_i at warning point b.

Definition 3 (Overflowable Path). *Given a warning $\omega = (\langle l_1, l_2, \ldots, l_n\rangle, b)$, with the warning path set $ps = (P_1, P_2, \ldots, P_m)$, $\forall P_i \in ps$, P_i is an overflowable path if* (2) *holds.*

$$\bigwedge_{j=1}^{j=n_i} \varphi_j^i \wedge OC(b) \equiv SAT \tag{2}$$

Definition 4 (Safe Path). *Given a warning $\omega = (\langle l_1, l_2, \ldots, l_n\rangle, b)$, with the warning path set $ps = (P_1, P_2, \ldots, P_m)$, $\forall P_i \in ps$, P_i is a safe path if* (3) *holds.*

$$\bigwedge_{j=1}^{j=n_i} \varphi_j^i \wedge OC(b) \equiv UNSAT \tag{3}$$

Definition 5 (Undecided Path). *Given a warning $\omega = (\langle l_1, l_2, \ldots, l_n\rangle, b)$, with the warning path set $ps = (P_1, P_2, \ldots, P_m)$, $\forall P_i \in ps$, P_i is an undecided path if* (4) *holds.*

$$\bigwedge_{j=1}^{j=n_i} \varphi_j^i \wedge OC(b) \equiv TIMEOUT \tag{4}$$

If we have found an overflowable path P_i in ps for warning ω, the warning ω will be regarded as a true warning. If we find a path that can validate the warning as a true warning, it's unnecessary to validate other paths in the warning path set ps. Otherwise, we need to traverse all paths in ps to validate a warning. If all paths in ps are safe paths, the warning ω will be regarded as a false warning. If some paths in ps are undecided paths, when there exists at least one overflowable path validating the warning as a true warning, the warning will be validated as a true warning regardless of the undecided paths. Otherwise, we cannot tell whether the warning is a false warning or not because there may be some overflowable paths hidden in the time-out paths. Therefore, we classify this kind of warnings as undecided warnings.

Definition 6 (True Buffer Overflow Warning). *Given a warning $\omega = (\langle l_1, l_2, \ldots, l_n\rangle, b)$, with the warning path set $ps = (P_1, P_2, \ldots, P_m)$, ω is a true buffer overflow warning if* (5) *holds.*

$$\exists P_i \in ps : \bigwedge_{j=1}^{j=n_i} \varphi_j^i \wedge OC(b) \equiv SAT \tag{5}$$

Definition 7 (False Buffer Overflow Warning). *Given a warning $\omega = (\langle l_1, l_2, \ldots, l_n\rangle, b)$, with the warning path set $ps = (P_1, P_2, \ldots, P_m)$, ω is a false buffer overflow warning if* (6) *holds.*

$$\forall P_i \in ps : \bigwedge_{j=1}^{j=n_i} \varphi_j^i \wedge OC(b) \equiv UNSAT \tag{6}$$

Definition 8 (Undecided Buffer Overflow Warning). *Given a warning $\omega = (\langle l_1, l_2, \ldots, l_n\rangle, b)$, with the warning path set $ps = (P_1, P_2, \ldots, P_m)$, ω is an undecided buffer overflow warning if it is neither a true buffer overflow warning nor a false buffer overflow warning, or, more formally, if* (7) *holds.*

$$\forall P_i \in ps : \bigwedge_{j=1}^{j=n_i} \varphi_j^i \wedge OC(b) \equiv UNSAT \qquad (7)$$

In some cases, all the paths in the warning path set ps are infeasible, namely $\bigwedge_{j=1}^{j=n_i} \varphi_j^i \equiv UNSAT$ for all paths, which will make (6) hold. Some of these false warnings can be found before symbolic execution, which will be discussed in Section 5. Other false warnings will be validated during symbolic execution, which will be discussed in Section 7.

4 Approach Overview

According to Fig.4, BovInspector consists of four modules: warning reachability analysis, guided symbolic execution, buffer overflow validation, and targeted automatic repair suggestions.

Warning Reachability Analysis: The input to this module is the set of buffer overflow warnings reported by static analysis tools together with the source code. For each buffer overflow warning w, this module computes all the complete paths that start from the program entrance, go through the statements in w, and end at the warning point b. For the warnings that are not reachable, this module can identify and prune them; the corresponding warnings are then classified as false warnings. This module will output the warning path set ps for each static buffer overflow warning w as guided information. Taking the warning ω_2=($\langle 19, 23, 9, 12\rangle, 14$) for the program in Fig.3 as an example, its warning path set is ps_2=($\langle 19, 23, 8, 12, 14\rangle$).

Guided Symbolic Execution: This module takes as input the guided information generated by the above module. It is an extension of the traditional symbolic execution engine. We perform symbolic execution on the source code starting from the program entrance. At each branching point with k possible branches, we first make k replicates of the execution state; Each replicate of the execution state represents a complete path from the program entrance to the branching point. At each branching point, this module tries to prune the state that cannot lead to the warning points by querying the guided information. When the execution reaches a warning point, we call the buffer overflow validation module to check whether buffer overflow can indeed happen.

Buffer Overflow Validation: This module takes as input the static buffer overflow warnings and outputs the validated buffer overflow warnings. During symbolic execution, each execution state maintains the necessary information, in particular path constraints. When the execution encounters a warning point, we will construct the buffer overflow constraints according to Table 1. Then the conjunction of the path constraints and the buffer overflow constraints will be fed to a constraint solver to examine whether all these conditions can be simultaneously satisfied. Based on the solution of constraint solving, we validate the path as an overflowable path, a safe path or an undecided path. If it is an overflowable path, then the warning will be validated as a true warning. A test case will be generated for it. Then the warning point will be removed from the checking list. Otherwise, the execution will continue until all the paths for the warning being executed. If all paths for a warning are safe paths, then the warning is a false warning. If there exist some undecided paths besides the false paths, the warning will be regarded as an undecided warning.

Targeted Automatic Repair Suggestions: By investigating 100 highly ranked buffer overflow vulnerabilities from 2009 to 2014 in the Common Vulnerabilities and Exposures (CVE)[5] and the benchmarks from

[5] `https://cve.mitre.org/`

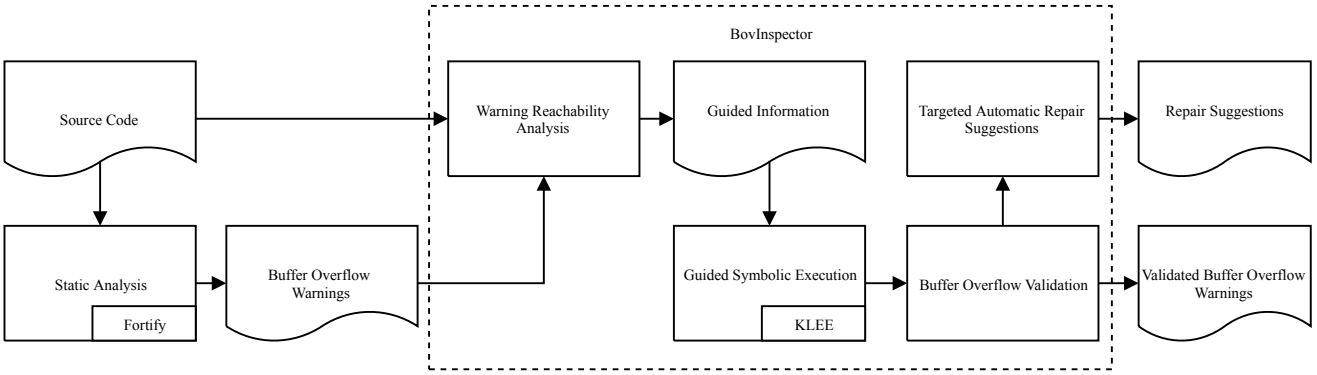

Fig.4. Overview of our automatic buffer overflow warning validation and bug repair approach

prior buffer overflow detection work, we [21] discover a total of 11 common repair strategies. For each validated true buffer overflow warning, we automatically generate repair code as suggestions for all the usable repair strategies. Moreover, programmers can also manually configure a preferred repair strategy.

5 Warning Reachability Analysis

For warning reachability analysis, we first generate the Interprocedural Control Flow Graph (ICFG) from the source code using the open source tool LLVM[6]. A Control Flow Graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths. A basic block (bb) is a linear sequence of program instructions that has one entry point (the first instruction executed) and one exit point (after the last instruction executed). ICFG considers the calling relationships among procedures and illustrates the control flow behavior of the whole program.

Fig.5 shows the ICFG of the example program in Fig.3. Each basic block is represented in the form of `fun.bb:line`, where `fun` is a function name, `bb` is the name of a basic block and `line` denotes the line number of the first statement in `bb`. In the following context, we identify the basic blocks of the program in Fig.3 by the line number of their first statement.

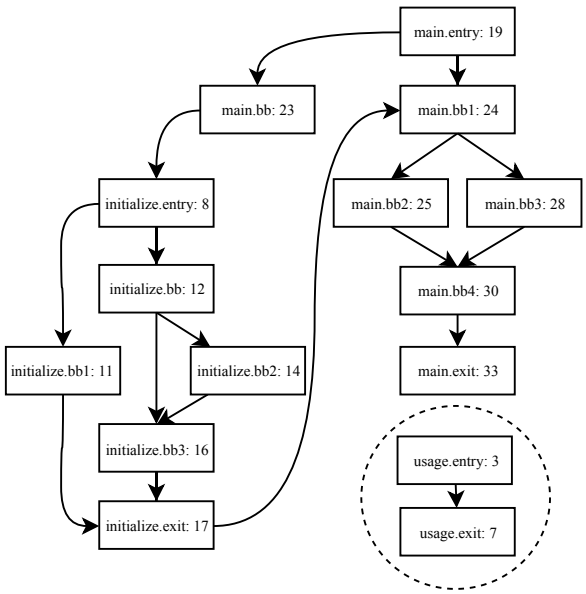

Fig.5. ICFG of the Example

After generating the ICFG for the given program, for each static buffer overflow warning $\omega = (\langle l_1, l_2, \ldots, l_n \rangle, b)$, we calculate all the complete execution paths that start from the program entrance, go through the statements contained in the warning $(l_1, l_2, \ldots, l_n)$, and end at the warning point b. To do that, we first map each statement label in ω[7] to its corresponding basic block, i.e., $l_1, l_2, \ldots, l_n, b$ cor-

[6] `https://llvm.org/`

[7] If there are several suspicious paths for one warning point, for presentation purpose, these paths will be split into several warnings. Next, for each warning, we compute its all possible paths. Then for each warning point, we join all the possible paths to construct its warning path set.

responds to $bb_{l_1}, bb_{l_2}, \ldots, bb_{l_n}, bb_{l_{n+1}}$. Then, we calculate the complete paths covering these basic blocks. A straightforward solution is to perform a depth-first traversal on the ICFG to find such complete paths. However, this will lead to the redundant traversal of many paths that do not contain any path segments of the buffer overflow warnings. In this work, for each warning point, we perform backward tracking on the ICFG starting from the warning point. By backward tracking, we can ignore the ICFG nodes that do not lead to the warning point. For each pair of ($bb_{l_{i-1}}$,bb_{l_i}), where $2 \leq i \leq n+1$, we use depth first search to calculate all the path segments between the two basic blocks (denoted as $PS_{i-1,i}$). Similarly, we analyze the partial warning path set $PS_{i-2,i-1}$ for ($bb_{l_{i-2}}$,$bb_{l_{i-1}}$). By combining and connecting these two partial warning path sets $PS_{i-2,i-1}$ and $PS_{i-1,i}$, we compute the partial warning path set $PS_{i-2,i}$ for ($bb_{l_{i-2}}$,bb_{l_i}). The above steps repeat until we reach the program entrance. Finally, we get the warning path set ps of a static buffer overflow warning ω.

Example. Considering the warning ω_2=$(\langle 19, 23, 9, 12\rangle, 14)$ for the example program in Fig.3, we calculate the complete execution paths that start from the program entrance (i.e., line `19`), end at the warning point (i.e., line `14`), and also contain lines `23`, `9`, and `12`. Line `14` in the source code corresponds to node `initialize.bb2:14` in the ICFG, line 12 corresponds to node `initialize.bb:12`, line 9 corresponds to node `initialize.entry:8`, line 23 corresponds to node `main.bb:23` and line 19 corresponds to node `main.entry:19` in the ICFG in Fig.5.

By backward tracking from `initialize.bb2:14` to `initialize.bb:12`, we get a partial warning path set $(\langle 12, 14\rangle)$. Similarly, we get the partial warning path set $(\langle 8, 12\rangle)$ from `initialize.bb:12` to `initialize.entry:8`, $(\langle 23, 8\rangle)$ from `initialize.entry:8` to `main.bb:23` and $(\langle 19, 23\rangle)$ from `main.bb:23` to `main.entry:19`. We thus calculate the warning path set ps_2=$(\langle 19, 23, 8, 12, 14\rangle)$ that covers the buffer overflow warning ω_2=$(\langle 19, 23, 9, 12\rangle, 14)$.

By warning reachability analysis, we can also identify some false warnings in ICFG. If a buffer overflow warning point cannot be reached by traversing ICFG from the program entrance, the buffer overflow warning is unreachable. In that case, such buffer overflow warnings are considered as false warnings. In our example program, the buffer overflow warning ω_0=$(\langle 4\rangle, 6)$ is unreachable as line `6` and line `4` in the source code correspond to node `usage.entry:3` and `usage.exit:7`. Node `usage.entry:3` is unreachable from the program entrance node `main.entry:19`, so it is a false warning.

The warning path set is the output of our warning reachability analysis. It records all complete paths of the buffer overflow warnings that can be reached from program entrance in the ICFG. The warning path set will be used as guided information in the guided symbolic execution.

6 Guided Symbolic Execution

In this work, the program under test is symbolically executed to identify whether the possible buffer overflows can be triggered.

As shown in Algorithm 1, we extend the traditional symbolic execution engine to support guided symbolic execution and buffer overflow validation. The extended parts are written in bold. To be noticed, a symbolic execution engine is quite complex and here we only excerpt the actions related to our algorithm. This module takes as input the static buffer overflow warnings reported by some static analysis tool and the warning path sets generated by warning reachability analysis, denoted as $WarningList$ and $Pathsets$. An $ExecutionStatePool$

will be maintained during execution. The overall process of guided symbolic execution is executing a loop until the pool is empty or time out (lines 6-10). In the loop, the pool will be updated by adding or removing states (line 9). This is the default move of the traditional symbolic execution engine. In each iteration, one of the states will be selected (line 7) to execute (line 8) according to a specific rule (e.g., randomly choose one). The *pc* (i.e., program counter) in the state represents the instruction to be executed. Before executing the instruction, we will first check its type. If the statement label of *pc* reaches a warning point (i.e., $\omega.b$), the buffer overflow validation module will be invoked (lines 13-14), which will be discussed in the next section. If it is a branch or loop instruction, we will fork a duplicate of the current state (i.e., *es*) and add it (i.e., *es*2) into the *AddedStateSet* (lines 15-19). Then *StatePruning* will be invoked to try to prune states that cannot lead to the warning points.

Here we try to prune states right after the symbolic execution engine finished interpreting a branch instruction and only check the two execution states that correspond to the branch. This strategy has two motivations. First, if the unnecessary state can be pruned at the very beginning when it is just generated, the symbolic execution engine can explore the necessary paths more efficiently. Second, the branch information can help us to identify an execution path better. That is because we use the statement label of the first statement of a basic block to represent the block in our warning path set, referring to Definition 2 in Section 3. After a branch, the *pc* of the true or false branch state is exactly the first statement of the corresponding block, i.e., $l1$ for *es* and $l2$ for *es*2.

Algorithm 1: Guided Symbolic Execution

```
   Input: WarningList, Pathsets
   Output: ValidationReport
 1 GuidedSymbolicExecution(){
 2 ExecutionStatePool = ∅;
 3 AddedStateSet = ∅;
 4 RemoveStateSet = ∅;
 5 ExecutionStatePool.add(initialState);
 6 while ExecutionStatePool.size > 0 && ! TIMEOUT do
 7     es = selectState(ExecutionStatePool);
 8     ExecuteInstruction(es);
 9     UpdateState(ExecutionStatePool, AddedStateSet,
         RemoveStateSet);
10 }
11
12 ExecuteInstruction(ExecutionState es){
13 if ∃ ω ∈ WarningList && es.pc == ω.b then
14     Call Validation Module;
15 if es.pc.instructionType == FORK then
16     es2 = fork(es);
17     es.pc = trueBranchStmt;
18     es2.pc = falseBranchStmt;
19     AddedStateSet.add(es2);
20     StatePruning(es, es2);
21 Other operations in traditional symbolic execution...
22 }
23
24 StatePruning(ExecutionState es, ExecutionState
     es2){
25 if es != NULL && es2 != NULL then
26     l1 = es.pc.stmtLabel;
27     l2 = es2.pc.stmtLabel;
28 if Pathsets.contains(l1) && !Pathsets.contains(l2) then
29     RemoveStateSet.add(es2);
30 if !Pathsets.contains(l1) && Pathsets.contains(l2) then
31     RemoveStateSet.add(es);
32 }
```

In *StatePruning*, to decide whether an execution state can be removed from the execution state pool, we need to check whether the execution state matches the warning path sets. Namely, we will check the existence of the two statement labels (i.e., $l1$ and $l2$) in *Pathsets* (lines 28-31). If only one statement label is contained in a certain warning path set, the execution state corresponding to the other statement label will be removed. If both statement labels are contained in the path set, we do not remove any of the execution states. Note that if both statement labels are not contained in the path set, we also do not remove any of the execution states. One reason is that the symbolic execution engine may explore the internal part of some library calls that do not appear in any warning path. Another rea-

son is to support validating the buffer overflow warnings that the warning point is in a **loop**. Because the paths in a warning path set start from the program entrance and end at the warning point. Namely, the branch information after the warning point is not contained in any warning path set. But the buffer overflow may be triggered by iterating the loop. Therefore, to iterate the loop, the two states of the branch after the warning point should not be pruned. In this way, our approach allows an execution to continue after calling the buffer overflow validation module if the current path cannot trigger a buffer overflow but the warning point is in a loop. When an execution reaches the exit of the loop, which is a branch, the execution state jumps back to the entrance of the loop. Since the buffer overflow point is in the loop, the entrance of the loop is contained in the warning path set. The execution will find a match and enter the loop again. Thus, our method can guide symbolic execution to iterate the loop until the loop ends or a buffer overflow is found within a loop time threshold.

Example. Taking the program in Fig.3 as an example, we have the warning path set ps_1=($\langle$ 19,23,8,12$\rangle$) for the warning ω_1=($\langle 19, 23, 9\rangle, 12$), the warning path set ps_2=($\langle 19, 23, 8, 12, 14\rangle$) for the warning ω_2=($\langle 19, 23, 9, 12\rangle, 14$) and the warning path set ps_3=(①$\langle 19, 24, 25, 30\rangle$, ②$\langle 19, 23, \underline{8, 11}, 24, 25, 30\rangle$, ③$\langle 19, 23, \underline{8, 12, 16}, 24, 25, 30\rangle$, ④$\langle 19, 23, \underline{8, 12, 14, 16}, 24, 25, 30\rangle$) for the warning ω_3=($\langle 25\rangle, 30$). The symbolic execution begins to explore program execution states from the entrance of the main program at line `19`. The execution executes the statements step by step until it meets a branch statement in the program at line `22`. Then, the symbolic execution engine duplicates the execution, points the two executions to the true branch at line 23 and the false branch at line 24 of the branch statement, respectively, and prepares to add the new execution to the execution state pool. Both line `23` and `24` are in the warning path sets, i.e., `23` in ps_1, ps_2 and ps_3, `24` in ps_3. Therefore, no state will be removed at this branch statement. When encountering the branch statement at line `24`, since the false branch at line `28` is not contained in any warning path set but line `25` is, we remove the false branch state from the execution state pool. When encountering the instruction at line `12, 14 or 30`, we find out it is a warning point and will invoke the buffer overflow validation module.

7 Buffer Overflow Validation

When the symbolic execution engine encounters an instruction matching a warning point in the warning list, the buffer overflow validation module will be invoked (lines 12-13 in Algorithm 1). The instruction encountered at the warning point is a buffer operation, which may be the APIs listed in Table 1, e.g., a call to strcpy. To validate static buffer overflow warnings during symbolic execution, we extend the symbolic execution engine to monitor buffer operations, collect path constraints, construct overflow constraints and validate warnings during symbolic execution.

To validate a warning, first we need to fetch the buffer information from the symbolic execution engine. It provides a memory management model to handle the information of all variables, including the addresses, size, contents, etc. By analyzing the memory object, we obtain the information of the buffer to be checked, including the starting address of the buffer, the offset from the starting address, and the size of the buffer. The symbolic execution engine updates the content in the buffer whenever there is an operation on the buffer. Therefore, before executing the instruction at the warning point, it is easy to fetch the latest content of the buffer related to the instruction. However, when the data copied to the buffer is symbolic, the

length information is difficult to obtain. To address this issue, we design a model to represent the length of a symbolic string variable. Let *buf* be a symbolic string variable, $Len(buf)$ denotes the length of string *buf* in bytes, and '\0' denotes the end of a string. In our model, we apply the following constraints based on string length to judge buffer overflow: $Len(buf) = i$, where $\forall j \in [0, i), S_{buf}[j] \neq$ '\0' and $S_{buf}[i] =$ '\0'. To unify the validation model, we apply the definition on both symbolic strings and concrete strings. As mentioned before, we use $Size(buf)$ to represent the size of the buffer assigned to the parameter *buf*, which can be obtained from the memory management model in the symbolic execution engine. When calculating the length of buffer *buf*, if the first $Size(buf)$ bytes has no '\0', it means that the string in the buffer has no terminator. In this case, we consider $Len(buf)$ to be infinite. To be noticed, the above assumption is only for our self-defined $Len(buf)$ in this paper. It is different from *strlen*(), which is a traditional API in C program and re-implemented symbolically in KLEE.

Based on the above buffer information, i.e., the starting address, offset, size and content, we next construct the buffer overflow constraints according to Table 1. For example, in this module, the overflow constraints $Len(src) \geq Size(dest)$ for strcpy will be constructed as $\bigwedge_{j=0}^{j=Size(dest)-1} S_{src}[j] \neq$'\0'during symbolic execution. The buffer overflow constraints constructed here correspond to the function $OC(b)$ in the definitions in Section 3. Then, we feed the conjunction of the path constraints and overflow constraints to a constraint solver. Based on the solution of constraint solving, we validate the path as an overflowable path, a safe path or an undecided path according to Definition 3-5 in Section 3. If it is an overflowable path, then the warning will be validated as a true warning, referring to Definition 6 in Section 3. A test case will be generated for it. Then the warning point will be removed from the checking list. Otherwise, the execution will continue until all the paths for the warning being executed or an overflowable path found for the warning. If all paths for a warning are safe paths, then the warning is a false warning, referring to Definition 7 in Section 3. If there exist some undecided paths besides the false paths, the warning will be regarded as an undecided warning, referring to Definition 8 in Section 3.

In order to improve the efficiency of BovInspector and avoid getting stuck in a loop or a path, we discuss several details of buffer overflow validation as follows. First, if we have validated a warning as a true warning, we will stop exploring other paths for this buffer warning point and report a true warning along with a test case that can follow the execution path and trigger the buffer overflow. Second, if we validate the current path as a safe path, we will check whether the current warning point is in a **loop**, if so, such execution will be allowed to continue until it reaches the upper bound of the loop or the path is validated as an overflowable path within a loop time threshold, as discussed in Section 6. Last, we set a time limit for the execution of each path to a buffer overflow warning point. If the procedure exceeds the time bound, it means that the solver cannot determine whether there is a solution to the constraints.

Example. Considering the example program in Fig.3, three static buffer overflow warnings need to be validated in this module, i.e., ω_1=($\langle 19, 23, 9\rangle$, 12), ω_2=($\langle 19, 23, 9, 12\rangle$, 14), ω_3=($\langle 25\rangle$, 30). Our validation module will be invoked when the symbolic execution engine executes the statement at line `12`, `14` and `30`. For the warning path ps_1=($\langle 19, 23, 8, 12\rangle$) leading to the warning point at line `12`, we will collect the path constraint $strlen(argv[1]) < 4$, $argv\_string == argv[1]$ and $strlen(argv\_string) < 24$. This module con-

structs the buffer overflow constraint of *strcpy* according to $Len(argv\_string) \geq Size(mapped\_argv)$ in Table 1, i.e., $\bigwedge_{j=0}^{j=3} S_{argv\_string}[j] \neq$ '\0'. The conjunction of all above constraints will be constructed as symbolic constraints and solved by a solver. The above constraints are not satisfiable. Hence the warning path ps_1 is a safe path and the warning ω_1=($\langle 19, 23, 9\rangle, 12$) is then validated as a false warning. Similarly, the constraints constructed for the warning path ps_2=($\langle 19, 23, 8, 12, 14\rangle$) leading to the warning point at line `14` are $strlen(argv[1]) < 4 \wedge argv_string == argv[1] \wedge strlen(argv_string) < 24 \wedge argv_string[strlen(argv_string) - 1] != $ '-' $\wedge \bigwedge_{j=0}^{j=2} S_{mapped\_argv}[j] \neq$ '\0'. By solving them with a constraint solver, we find out that the constraints are satisfiable. Therefore, the warning path ps_2 is an overflowable path and the warning ω_2=($\langle 19, 23, 9, 12\rangle, 14$) is then validated as a true warning. Meanwhile, a test case `(argv[1] == "abc")` that satisfies the whole constraints will be generated to trigger the buffer overflow. Likewise, the warning ω_3=($\langle 25\rangle, 30$) is validated as a false warning.

8 Targeted Automatic Repair Suggestions

Fixing the validated buffer overflow defects in the program is an urgent task for the developers for the sake of security. It is time consuming to manually fix the existing defects. Therefore, using automatic techniques to supplement manual software development is becoming a trend. However, automatic software repair is challenging because it is a difficult task to figure out where the bug is and how to generate a programmer's preferred fix. Our buffer overflow validation module has solved the "where the bug is" problem. It reports the true buffer overflow warnings, each of which contains the information of the buffer APIs, the size of buffer, the locations (i.e., statement label) of buffer initialization, buffer operations, overflow triggering, and the test case to trigger the vulnerability and corresponding execution paths. With such information, we know "where the bug is". The next step is to figure out "how to fix the bug in a way that will be adopted by programmers". Automatic software repair is a promising way to reduce the manual work for programmers. The problem is that programmers tend to lack confidence in the code repaired by some automatic tools. Improving the understandability of the repaired code will be helpful for the programmers to validate and adopt the repair suggestions. Therefore, we performed some empirical studies to survey the officially adopted or programmers' preferred fix approaches for buffer overflow vulnerabilities. By investigating 100 highly ranked buffer overflow vulnerabilities from 2009 to 2014 in the Common Vulnerabilities and Exposures (CVE) and the benchmarks from prior buffer overflow detection work, we discovered a total of 11 common repair strategies [21]. The results show that nearly half of these vulnerabilities (48%) are patched by adding a bounds check before the buffer operations, while API replacement and using larger buffer share the second place. All the 11 kinds of repair strategies mentioned in [21] are shown in Table 2 in decreasing order of usage frequency. These strategies fix buffer overflow vulnerabilities by adding checks or smashing overflow conditions. Inspired by these official or programmers' preferred repair habits, we adopt these 11 types of repair strategies as repair templates, assemble the final repair codes using the contexts of the buffer overflow vulnerabilities, and provide repair suggestions total automatically or according to the repair mode selected by the developers.

The specific steps of targeted automatic repair suggestions are shown in Algorithm 2. This module takes as input the source code of the target program, the true buffer overflow warnings validated by BovInspec-

Table 2. Buffer Overflow Repair Templates

Rank	Repair Mode	Repair Strategies	Templates	Allowed Context
1	ABC	Adding Boundary Check	if(Overflow Constraints) exit(error);	Any context allowed
2	AR	API Replacement	$snprintf(dest, Size(dest), format, ...)$	$strcpy/sprintf/strcat/vsprintf/scanf/vscanf/sscanf/vsscanf/fscanf/vfscanf$
2	ULB	Using Larger Buffer	static buffer: TYPE $buf[newSize]$; dynamic buffer: buf=$realloc(buf, newSize)$;	static buffer: TYPE $buf[oldSize]$; dynamic buffer: TYPE* buf;
4	FBC	Fixing Boundary Check	if(Overflow Constraints){...}	Boundary Check exists.
5	AIC	Adding Integer Check	if $(i + j < i \mid\mid i + j < j)$ exit(error);	$buf[i + j] = x$;
6	ASE	Adding String End	if$(i >= Size(buf))$ $dest[Size(buf) - 1]$='\0';	$buf[i] = x$;
7	AMC	Adding Malloc Check	if(!buf) exit(error);	$buf = malloc(n)$;
8	SR	String Reformat	$snprintf(str, Size(str)$, "...%%%ds ...", ... , $Size(dest)$-1,...); $sscanf(src,$ "...%s...", ...$dest$...);	$sscanf(src,$ "...%s...", ...$dest$...)
9	UUV	Using Unsigned Value	unsigned i;	Boundary check $i >= 0$ is needed.
10	LIR	Limiting Index Range	$i = MIN(i, Size(dest)/sizeof(*dest) - 1)$; $i = MAX(i, 0)$;	$buf[i] = x$;
11	UCL	Using Concrete Length	$bufAPI(dest, src, ConcreteValue)$;	$bufAPI(dest, src, n)$;

Note: For ULB, the newSize of static buffer is configured by programmers or other static size analysis tools and is determined at compile time; the newSize of dynamic buffer is computed based on the overflow constraints at runtime.

Table 3. Buffer Overflow Repair Solution

Type	API	ABC	AR	ULB	FBC	AIC	ASE	AMC	SR	UUV	LIR	UCL
Unbounded content sensitive buffer operations	*strcpy*, *sprintf* *strcat*, *vsprintf*	✓	✓	✓	✓			✓				
	scanf, *sscanf*, *fscanf* *vscanf*, *vsscanf*, *vfscanf*	✓	✓	✓	✓			✓	✓			
Bounded content sensitive buffer operations	*strncpy*, *snprintf*, *fgets* *vsnprintf*, *fread*, *read*	✓		✓	✓			✓			✓	✓
	strncat	✓		✓	✓			✓				✓
Bounded content insensitive buffer operations	*memcpy* *memmove* *memset*	✓		✓	✓			✓			✓	✓
Direct buffer accesses	$buf[i]$, $*(buf + i)$	✓		✓	✓	✓	✓	✓		✓	✓	

tor, and a repair mode configured by programmers, which can be empty. The repair may insert some statements in the code, which will result in a line number mismatch between the source code and the report of the other warnings. To keep the consistency of line numbers by handling true warnings from the bottom to the top, we first sort all the true warnings in each source file by the line numbers in descending order. For each validated true buffer overflow warning, we extract the necessary information for repair, such as buffer API, buffer size, definition location of the buffer, and location of the buffer API, etc. We first query the usable repair modes for the buffer API in Table 3, marked as *UsableRepairModeList* in line 4. If the programmer provides a none empty repair mode and the mode is usable, we clear the modes in *UsableRepairModeList* and use the configured repair mode instead (lines 5-7). Otherwise, we use the modes

in the original *UsableRepairModeList* computed at line 4. For each mode in *UsableRepairModeList*, we first check whether the precondition is satisfied by analyzing the context of the buffer operation in the program. The allowed contexts are shown in the last column in Table 2. For repair mode *SR*, only sscanf is listed as an example in Table 2. Other APIs, such as *scanf*, *vscanf*, *sscanf*, *vsscanf*, *fscanf* and *vfscanf*, use the corresponding allowed contexts and templates. The allowed context to use *SR* to repair sscanf is: the buffer API sscanf is storing characters to the buffer *dest* with a string format like "%s". If the allowed context is satisfied, we then query the template for the repair mode. Next, we assemble the repair code based on the template and the contexts of the buffer API in the source code. For repair mode *SR*, the core idea is to control how many bytes are written into a buffer by a formatting string. *SR* will convert the string format "%s" in sscanf into "%ns" by applying an *snprintf* operation, where n is $Size(dest) - 1$.

Algorithm 2: Targeted Automatic Repair Suggestions

Input: SrcCode, TrueWarningSet, RepairMode
Output: RepairedCodeSuggestions

```
1  SortedTrueWarningSet = Sort(TrueWarningSet);
2  foreach w ∈ SortedTrueWarningSet do
3      bufAPI, bufSize, bufLoc, bufAPILoc =
         ExtractInfo(w);
4      UsableRepairModeList =
         QueryRepairSolutions(bufAPI);
5      if RepairMode != EMPTY && RepairMode ∈
         UsableRepairModeList then
6          UsableRepairModeList.clear();
7          UsableRepairModeList.add(RepairMode);
8      foreach mode ∈ UsableRepairModeList do
9          isSatisfied =
             CheckPrecondition(mode, bufAPILoc);
10         if isSatisfied then
11             template = QueryRepairTemplate(mode);
12             code = AssembleRepairCode(SrcCode,
                 template, bufAPI, bufSize, bufLoc,
                 bufAPILoc);
13             RepairedCodeSuggestions.add(code);
14         else
15             ReportError(w, mode);
```

To be noticed, when using BovInspector to repair a program, *sizeof(buf)* is used instead of the value of *Size(buf)* when the statement in function f is accessing a buffer buf that is a locally defined (namely defined in the same function f) static array or a globally defined static array. Because by analyzing official repairs, we found "sizeof" is commonly used in such cases. When the above conditions are not satisfied, we will retrieve the size of the buffer from the output of BovInspector, i.e., $Size(buf)$. In the body of the new `if` statement added by BovInspector, we take "return 0;", "return false;" or "return NULL;" as the return statement depending on the return type of the function. The return statement we added may be incorrect, analyzing a more precise return statement based on the program context is left as future work.

This module provides the corresponding repaired codes for each usable repair mode and ranks them in the order of the repair modes listed in Table 2, namely in decreasing order of usage frequency. Programmers can simply take the first one as the repaired code, which is the most commonly used repair mode in official repairs, or they can configure a preferred repair mode. Note that these repair methods only ensure that the buffer overflow will not happen for the current line of code. We recommend users to manually decide whether to adopt the repair suggestions.

Table 4. Safe API Used In Repair Mode AR

API	Safe API Option
strcpy	$strncpy(dest, src, Size(dest) - 1)$ $snprintf(dest, Size(dest), \text{"\%s"}, src)$
strcat	$snprintf(dest + strlen(dest), Size(dest) - strlen(dest), \text{"\%s"}, src)$
sprintf,*vsprintf*	$snprintf(str, Size(str), format, ...)$
scanf, *vscanf*	*scanf_s*, *vscanf_s*
sscanf, *vsscanf*	*sscanf_s*, *vsscanf_s*
fscanf, *vfscanf*	*fscanf_s*, *vfscanf_s*

Example. To illustrate the effect of the targeted automatic repair suggestions module, we take the vali-

```
12 //Original programs
13 if (argv_string[0] != '-') {
14     strcat(mapped_argv, "-");
15 }
```

```
12 //Repaired programs
13 if (argv_string[0] != '-') {
14     if (strlen(mapped_argv)+strlen("-")
                >= sizeof(mapped_argv)){
15         fprintf(stderr, "error,exit\n");
16         return 1;
17     }
18     strcat(mapped_argv,"-");
19 }
```

Mode: ABC (Adding Boundary Check)

```
13 //Original programs
14 strcat(mapped_argv, "-");
```

```
13 //Repaired programs
14 snprintf(mapped_argv+strlen(mapped_argv),
     sizeof(mapped_argv)-strlen(mapped_argv),
     "%s", "-");
```

Mode: AR (API Replacement) using snprintf

```
8 //Original programs
9 char mapped_argv[MIN_LEN];
```

```
8 //Repaired programs
9 char mapped_argv[5];
```

Mode: ULB (Using Larger Buffer)

Fig.6. Repair Result for Example Program

dated buffer overflow in Fig.3 as an example. As mentioned in Section 7, BovInspector validates the *strcat* at line 14 as a true buffer overflow and generates a buffer overflow report for it. The report is then used as an input to the targeted automatic repair module. According to Table 3, there are 5 usable repair modes for buffer API *strcat*, namely *ABC, AR, ULB, FBC and AMC*. But only *ABC, AR and ULB* are actually usable after checking the allowed context in Table 2. Repair mode *AMC* is not usable because no malloc instruction exists for buffer *mapped_argv*. Fig.6 shows the corresponding repaired source code of the vulnerability by applying the usable repair modes *ABC, AR and ULB*. For repair mode *ABC*, we fetch the parameters *mapped_argv* and `"-"` from the operation of *strcat*, compute the $OverflowConstraints$ according to Table 1. The condition inserted is "`if(`$strlen$`(mapped_argv) +` $strlen$`("-") >= sizeof(mapped_argv)`". The code segment is instrumented in front of the buffer operation in the source code to ensure that the buffer operation will not be executed if the overflow constraint is satisfied. Thus, the buffer overflow will not be triggered. For repair mode *AR*, *strcat* can be replaced by its safe API *snprintf* with appropriate parameters according to Table 4. For repair mode *ULB*, we first perform a static analysis [22] to compute the length range of the characters in *mapped_argv* at line 14, which is $(1, 4]$. Therefore, we modify the size of *mapped_argv* to 5 at line 9. If the length of the string cannot be calculated, we let the programmers configure the new buffer size or just use n times the old size, where n is configured by programmers (2 by default).

9 Experimental Results

Implementation: We implemented BovInspector[8] by extending the tool proposed in our previous work [18]. The tool is based on LLVM 2.9[9] and KLEE [20]. We use the commercial Fortify 3.2 as the static buffer overflow detector.

[8]BovInspector is available at `http://bovinspectortool1.github.io/project/`

[9] `https://releases.llvm.org/2.9/docs/ReleaseNotes.html`

Evaluation Goals: For evaluation, we hope to answer the following key research questions:

RQ1: Whether our validation technique is effective and precise for classifying static buffer overflow warnings?

RQ2: Whether the performance and scalability of our technique is acceptable on real-world applications?

RQ3: Whether our automatic repair technique is effective?

Metrics: The symbols used in the metrics are defined in Table 5.

Table 5. The Symbols Used in the Metrics

Symbol	Meaning
$\#L$	The number of lines of code
$\#W$	The number of buffer overflow warnings
$\#P$	The number of buffer overflow warning paths
$\#true$	The number of true warnings validated by BovInspector
$\#false$	The number of false warnings validated by BovInspector
$\#undecided$	The number of undecided warnings that cannot be validated by BovInspector
$\#FP$	The number of false positives
$\#fixed$	The number of warnings that will not be reported by the static analysis tool Fortify after being repaired by BovInspector

For RQ1, we first define *DCR* and *UCR* to evaluate the effectiveness of our approach.

Decidable Classifying Ratio (*DCR*) of warning validation is the percentage of warnings that are validated as true or false warnings by BovInspector in all the static buffer overflow warnings.

$$DCR = \frac{\#true + \#false}{\#W}$$

Likewise, *DCR* of warning path validation is the percentage of warning paths that are validated by BovInspector.

Undecidable Classifying Ratio (*UCR*) for warning validation is the percentage of warnings that are undecided warnings that cannot be validated by BovInspector in all the static buffer overflow warnings.

$$UCR = \frac{\#undecided}{\#W}$$

Likewise, *UCR* of warning path validation is the percentage of warning paths that cannot be validated by BovInspector.

For RQ2, we use time consumptions and memory consumptions to measure the performance of our approach in the three stages of reachability analysis, symbolic execution, and buffer overflow validation.

For RQ3, recall ratio of static analysis (RRSA) is used to measure the percentage of unfixed warnings in the validated true buffer overflow warnings.

$$RRSA = \frac{\#true - \#fixed}{\#true}$$

Experimental Setup: To prepare the benchmark, we selected 8 programs from GNU COREUTILS utilities[10] and real-world open source programs such as sendmail-8.12.7[11]. All experiments were conducted on a quad-core machine with an Intel Core (TM) Corei5-2400 3.10GHz processor and 4G memory, running Linux 3.11.0.

9.1 RQ1: Effectiveness and Precision of the Validation of Static Buffer Overflow Warnings by BovInspector

To answer RQ1, we performed experiments on both synthetic and real-world programs. Table 6 shows the synthetic programs used in the experiments, which are eight programs from GNU COREUTILS utilities. These programs are relatively small and we can manually check the results.

To increase the number of static analysis warnings for each program, we manually inserted or removed

[10] `https://www.gnu.org/software/coreutils/`

[11] `http://www.sendmail.org/~ca/email/sm-812.html`

Table 6. Buffer Overflow Validation Result on Synthetic Programs

Program	BovInspector Input			BovInspector Output									
				Warning Points					Warning Paths				
	$\#L$	$\#W$	$\#P$	$\#true$	$\#FP$	$\#false$	$\#FP$	$DCR(\%)$	$\#true$	$\#FP$	$\#false$	$\#FP$	$DCR(\%)$
chmod	0.6k	12	12	7	0	4	0	92	7	0	4	0	92
tr	1.9k	8	15	3	0	3	0	75	6	0	5	0	73
pwd	0.4k	12	16	3	0	7	0	83	3	0	10	0	81
sort	3.3k	22	27	8	0	7	0	68	10	0	9	0	70
su	0.6k	6	8	1	0	3	0	67	2	0	3	0	63
ls	4.6k	31	42	4	0	19	0	74	10	0	22	0	76
pr	2.9k	29	31	11	0	15	0	90	13	0	15	0	90
df	1.0k	18	18	9	0	9	0	100	9	0	9	0	100

Table 7. Examples of Synthetic Program Generation

Type	Original Code	True Warning Code	False Warning Code

Unbounded content sensitive buffer operations	if($strlen(src) >= Size(dest)$)
	  return;
	$strcpy(dest, src)$;
		$strcpy(dest, src)$;	unsigned $m, n1, n2$;
			$scanf(\text{“}\%d\%d\text{”}, \&m, \&n1)$;
			$n2 = n1 * m$;
			if($n2$ <100 && $n2$ >0 && m >100)
			  $strcpy(dest, src)$;

Bounded content sensitive buffer operations	$strncpy(dest, src, size(dest))$;	$strncpy(dest, src, Size(dest) + 1)$;	if(false){
			  $strncpy(dest, src, Size(dest) + 1)$;
			}

Bounded content insensitive buffer operations	$memcpy(dest, src, Size(dest))$;	$memcpy(dest, src, size(dest) + 1)$;	if(false){
			  $memcpy(dest, src, Size(dest) + 1)$;
			}

Direct buffer accesses	if((i+1)*sizeof(*buf)<=$Size(buf)$) $buf[i]$;	$buf[i]$;	if(false){ $buf[i]$; }

Note: “if(false)” represents an infeasible path, referring to the false warning code for strcpy.

code snippets in several random positions in the program. These code snippets will make Fortify report buffer overflow warnings, including both true and false buffer overflow warnings. True buffer overflow warnings are introduced by removing the existing boundary checks for insecure buffer manipulations. False buffer overflow warnings are introduced by inserting buffer manipulations with infeasible paths. Table 7 shows the examples of how to generate synthetic programs. For unbounded content sensitive buffer operations, we take *strcpy* for example. To insert true warnings, we randomly located several code snippets in the program matching the original code in the second row in Table 7 and manually check whether it is a true warning if we remove the boundary check for each location. If so, a synthetic true warning is generated by removing the boundary check in each selected location. Similarly, for direct buffer accesses, true warnings are also injected by removing the boundary check. For bounded content sensitive and insensitive buffer operations, we take *strncpy* and *memcpy* for example. True warnings are inserted by changing the third parameter from “$size(dest)$” to “$size(dest) + 1$”, i.e., creating an off-by-one buffer overflow. To insert false warnings, we first constructed true warnings and then added infeasible path constraints to the true warn-

Table 8. Buffer Overflow Validation Result on Real Programs

Program	BovInspector Input			BovInspector Output									
				Warning Points					Warning Paths				
	$\#L$	$\#W$	$\#P$	$\#true$	$\#FP$	$\#false$	$\#FP$	$DCR(\%)$	$\#true$	$\#FP$	$\#false$	$\#FP$	$DCR(\%)$
polymorph-0.40	0.3k	11	19	0	0	11	0	100	0	0	19	0	100
bc1.06	9.7k	5	15	1	0	3	0	80	1	0	11	0	80
net-tool1.46	8.1k	62	1256	6	0	36	0	68	251	0	763	0	81
wwwcount2.3	8.3k	20	112	1	0	5	0	30	1	0	50	0	45
gzip1.2.4	5.1k	19	237	1	0	12	0	68	4	0	148	0	64
sendmail8.12.7	78.1k	96	4854	1	0	51	0	54	1	0	3067	0	63

Table 9. Buffer Overflow Details in Real Programs

Program	API	Buffer API Location	Reference	Official Repair Version
polymorph-0.40	*strcpy*	polymorph.c:118	EDB-ID:22633	ϕ
bc-1.06	array access	main.c 188	[7]	ϕ
net-tools-1.46	*strcat*	netstat.c:450,457,602,608,737,743	N/A	ϕ
wwwcount-2.3	*strcpy*	parse.c:840	CVE-1999-0021	wwwcount-2.5
gzip-1.2.4	*strcpy*	gzip.c:1009	CVE-2001-1228	gzip-1.3.9
sendmail-8.12.7	array access	headers.c:1337-1342	CVE-2002-1337	sendmail-8.12.8

Note: ϕ means we have not found an official repair for the buffer overflow. N/A means the reference of the report of the buffer overflow defect is unavailable.

ings. For example, the path constraints constructed for $strcpy(dest, src)$ in the fourth column in Table 7 are $n1, m \in \mathbb{N} \wedge n1 \times m < 100 \wedge n1 \times m > 0 \wedge m > 100$, which are unsatisfiable.

Table 8 shows the six real-world programs used in the experiments. The six programs were used in previous buffer overflow detection work [7, 13, 23–25]. Real-world buffer overflows in each program are located by comparing the current version with the repaired version, reading the log of the repaired version and referring to the CVE report[12]. The details of the real buffer overflows are shown in Table 9.

Table 6 shows the results on synthetic programs and Table 8 shows the results on real programs. We show the basic information of BovInspector in the first four columns, i.e., $\#L$, $\#W$ and $\#P$. From the fifth column to the fourteenth column, we show the result for warning points and warning paths. For each of them, we show five types of output of BovInspector, i.e., $\#true$ and its $\#FP$, $\#false$ and its $\#FP$ and DCR.

Result 1: BovInspector can significantly reduce the number of buffer overflow warnings to be manually checked due to its ability of validating static buffer overflow warnings. From Table 6 and Table 8, we can find that BovInspector can effectively validate most of the warning points and the warning paths. For warning points, the DCR on the synthetic programs ranges from 67% to 100% and the DCR for all the synthetic programs together is 82%; the DCR on the real programs ranges from 30% to 100% and the DCR for all the real programs together is 60%. For warning paths, the DCR on the synthetic programs ranges from 63% to 100% with an average of 81%; the DCR on the real programs ranges from 45% to 100% with an average of 67%. In general, the UCR on real-world programs is 40%. In other words, there are about

[12] `https://cve.mitre.org/`

40% of static warnings, namely undecided static warnings, that need to be manually checked by programmers in real-world programs. It means BovInspector can significantly reduce the number of buffer overflow warnings to be manually checked.

Result 2: In practice, the true warnings and false warnings identified by BovInspector are all correct. For the synthetic programs in Table 6, we manually examined all the buffer overflow warnings reported by Fortify, recorded the verified buffer overflow list and compared them with the true buffer overflow warnings and false buffer overflow warnings validated by BovInspector. As shown in Table 6, column $\#FP$ records the number of false positives for the true warnings and false warnings validated by BovInspector, respectively. The values in column $\#FP$ are all 0, which means the true warnings and false warnings identified by BovInspector are all correct. Moreover, there is a one-to-one correspondence between the manually verified buffer overflow list and the warnings validated by BovInspector. By manually validating the warning paths, we found that the true and false warning paths validated by BovInspector are all correct. For the real-world programs in Table 8, after validation, BovInspector will label the warning paths as overflowable paths or safe paths (denoted as $\#true$ and $\#false$ in the "Warning Paths" column) for each warning. We manually checked all the overflowable ($\#true$) paths reported by BovInspector. For the safe ($\#false$) paths reported by BovInspector, we only randomly checked 200 of them or all of them if less than 200 by hand. It shows that there is no false positive for the validation of warning paths and warning points by BovInspector. Furthermore, all the true buffer overflows validated by BovInspector are included in the real buffer overflow lists, referring to Table 9. We can also see that false warnings account for about 92.2 percent of all the decided warnings. The experiments on the synthetic programs and the real programs show that the result of our method for decided warnings is reliable.

9.2 RQ2: Performance and Scalability of BovInspector

To evaluate the performance and scalability of our method on real-world programs, we recorded the time and memory consumptions of BovInspector on the six real-world programs used in Table 8. Moreover, to show the benefit of introducing the warning reachability analysis and guided symbolic execution, we conducted two sets of experiments shown in Table 10. For the first set, we listed the statistics of BovInspector. We first listed the statistics of the two stages of BovInspector, i.e., $\#RA$ for reachability analysis and $\#GSE + BV$ for guided symbolic execution with buffer overflow validation. We then took the sum of the time consumptions of $\#RA$ and $\#GSE + BV$ as the time consumptions of BovInspector and listed the values in column $\#All$. We recorded the peak memory consumption of $\#RA$ and $\#GSE + BV$ as the memory consumption of BovInspector and listed the values in column $\#Peak$. To be noticed, for the item $\#GSE + BV$, we only considered those validated true or false buffer overflow warnings, because the undecided buffer overflow warnings are limited by a time bound, which may disturb the accuracy of the statistics. Symbolic execution will be terminated when all the buffer overflow warning points and warning paths are traversed. For the second set, a comparison experiment was performed to study whether it is also capable of validating static warnings fast without the guidance of warning reachability analysis, namely the column labeled as "Unguided". In this case, we performed buffer overflow validation by using unguided symbolic execution. Namely, we omitted the warning reachability analysis and simply took as input the buffer

Table 10. Time and Memory Consumption on Real Programs

Program	Time Consumption(s)				Memory Consumption(MB)			
	BovInspector			Unguided	BovInspector			Unguided
	$\#RA$	$\#GSE+BV$	$\#All$		$\#RA$	$\#GSE+BV$	$\#Peak$	
polymorph-0.40	0.1	101.2	101.3	294.2	2.4	36.1	36.1	87.1
bc1.06	3.1	125.9	129.0	201.1	22.2	77.4	77.4	283.4
net-tool1.46	39.8	282.7	322.5	2547.7	68.0	394.5	394.5	512.3
wwwcount2.3	9.1	159.3	168.4	352.6	19.4	225.1	225.1	435.9
gzip1.2.4	16.7	169.2	185.9	453.7	15.1	64.2	64.2	166.2
sendmail8.12.7	145.8	5758.2	5904.0	7124.1	109.2	1435.1	1435.1	3046.9

overflow warnings reported by Fortify. Actually, to be fair, only the decided warnings validated by BovInspector will be fed to the symbolic execution engine in the "Unguided" experiment. Since no path information is available for the "Unguided" method, symbolic execution will be terminated when all the buffer overflow warning points are traversed. In that case, the "Unguided" method may miss a lot of executions to trigger buffer overflow. Therefore, the actual time and memory consumptions of the "Unguided" method would be even larger than the values shown in Table 10.

Result 3: Our method is capable of handling large-scale real-world programs. Table 8 shows the buffer overflow validation results on real programs. Most of the programs have almost 10 thousand lines of source code while the largest one reaches 78.1k. We observed that BovInspector performs well on most of these programs and validated a large number of false warnings, especially for the warning paths. About 60% of the static buffer overflow warnings are decidable for BovInspector. According to Table 10, BovInspector can finish validating all the decidable static buffer overflow warnings in these real-world programs with an acceptable time consumption and memory consumption.

Result 4: Warning reachability analysis and guided symbolic execution are effective for improving the performance and scalability of BovInspector. By comparing the statistics of BovInspector and the "Unguided" method, we found that the "Unguided" method always consumes more time and memory than BovInspector. The gap becomes more obvious especially when the scale of the program increases. This means the warning reachability analysis and guided symbolic execution perform well in saving time and memory by reducing the exploration space of symbolic execution. By comparing the time consumption of BovInspector and "Unguided", we can see that relying on the guidance of the warning reachability analysis can save about 17.1% to 87.3% of time, with an average of 37.9%. The result shows that BovInspector consumes about 23.0% to 72.7% with an average of 50.7% less memory than the "Unguided" method. Despite we only made a relatively conservative statistics on the "Unguided" method, namely the data for it would be even larger than the value shown in Table 10, BovInspector still costs much less time and memory than the "Unguided" method. In that case, we can conclude that applying warning reachability analysis and guided symbolic execution can effectively reduce the size of search space, which further significantly increases the performance and scalability of BovInspector.

Table 11. Validation and Repair Results on Real Programs

Program	API	#W	Fortify (before)	BovInspector	Fortify (after)			
					ABC	AR	ULB	LIR
polymorph-0.40	*strcpy*	1	Bov	Bov	Dangerous function	ϕ	N/A	N/A
bc-1.06	array access	1	Bov	Bov	ϕ	N/A	Bov	Bov
net-tool-1.46	*strcat*	6	Bov	Bov	ϕ	ϕ	N/A	N/A
wwwcount-2.3	*strcpy*	1	Bov	Bov	Dangerous function	ϕ	N/A	N/A
gzip-1.2.4	*strcpy*	1	Bov	Bov	Dangerous function	ϕ	N/A	N/A
sendmail-8.12.7	array access	1	Bov	Bov	ϕ	N/A	N/A	N/A

Note: Bov means buffer overflow. ϕ means there is no report from Fortify after applying the corresponding repair. N/A means the corresponding repair mode is unavailable for the bug.

9.3 RQ3: Effectiveness of Targeted Automatic Repair

After validating the buffer overflow warnings of the programs in Table 8, we continued to evaluate the effectiveness of the repair module of BovInspector on these programs. After repair, we used Fortify to re-scan the repaired code to see whether the buffer overflow is fixed. To study the reliability of the repair of BovInspector, we made a comparison between the repair codes generated by BovInspector and the official repairs. According to Table 9, there is no official repair version found for the first three programs. To further enrich the repair examples, we repaired another seven real-world programs with known buffer overflows using BovInspector.

Result 5: The targeted automatic repair module can fix buffer overflow in most cases. Table 11 shows the results of Fortify on the original programs and the programs repaired by BovInspector. From the column "Fortify (before)" and BovInspector, we can see that all the buffer overflow warnings are validated as true buffer overflow warnings. The column "Fortify (after)" lists the Fortify results for the four repair modes, i.e., ABC (adding boundary check), AR (API replacement), ULB (using larger buffer) and LIR (limiting index range). We observed that by applying the ABC repair mode of BovInspector to the tested programs, three buffer overflow warnings are degraded to dangerous function, which is a very low-risk warning in Fortify and the other eight buffer overflow warnings are eliminated from the report of Fortify. Therefore the *RRSA* in ABC mode is 27.3%. The AR mode performs better. All the nine buffer overflow warnings are not reported by Fortify after being repaired by BovInspector in AR mode and therefore the *RRSA* is 0%. All these repaired codes are verified to be correct by manual inspection. For the repaired code of bc-1.06, by applying the ULB and the LIR mode, Fortify still reports buffer overflow warning on it due to the bug finding schemes of Fortify. Through re-validation by BovInspector and manual inspection, we found that the warning for ULB is a true warning, but the warning for LIR is a false warning.

Result 6: Our repair method is similar to the human programmers' repair habit.

Table 12 shows the results of the BovInspector's repair and the official repair. The "BovInspector Repair" column shows the repair results for the usable repair modes. ABC and AR are usable for strcpy and sprintf. ABC and LIR are usable for array access. We listed all the repair results of the usable repair modes to further show how BovInspector repairs buffer overflow vulnerabilities. The last column shows the results of the official repairs. We also presented the repair modes used by official repair. As we can see, the code repaired by BovInspector, when using the same repair mode with the official repair, is very similar to the code repaired by

Table 12. Results of the BovInspector's Repair and the Official Repair

Program	Location	API	BovInspector Repair	Official Repair
wwwcount -2.3	parse.c 840	*strcpy*	ABC: if($strlen(qs)$ >= $sizeof(query\_string)$) return 1; AR: $strncpy(query\_string, qs, sizeof(query\_string)-1)$; $query\_string[sizeof(query\_string)-1]$='\0';	AR: $safeStrcpy(query\_string, qs, sizeof$ $(query\_string)$-1) ;
gzip -1.2.4	gzip.c 1009	*strcpy*	ABC: if($strlen(iname)$ >= $sizeof(ifname)$) return 1; AR: $strncpy(ifname, iname, sizeof(ifname)-1)$; $ifname[sizeof(ifname)-1]$='\0';	ABC: if($sizeof$ $ifname-1$<=$strlen(iname)$)
sendmail -8.12.7	headers.c 1337	array	ABC: if($strlen(bp)$>= (MAXNAME + 1)) return bp;	ABC: if($realqmode$ &&bp<$bufend$)
man -1.5i2	man.c 299	*strcpy*	ABC: if($strlen(name0)$>= $sizeof(ultname)$) return NULL; AR: $strncpy(ultname, name0, sizeof(ultname)-1)$; $ultname[sizeof(ultname)-1]$='\0';	ABC: if($strlen(name0)$>=$sizeof(ultname)$) {return $name0$;}
wu-ftpd -2.5.0	ftpd.c 1210	*strcpy*	ABC: if($strlen(mapped\_path)$ >= 1024) return NULL; AR: $strncpy(path, mapped\_path, 1023)$; $path$[1023]='\0';	AR: $strncpy(path, mapped\_path, size)$; $path[size-1]$ = '\0';
xmp -2.5.1	dtt_load.c 79	array	ABC: if($(i$+1)$*sizeof(*pofs)$>$sizeof(pofs)$) return 1; LIR: $i = MIN(i, sizeof(pofs)/sizeof(*pofs)-1)$;	ABC: if (i < 256)
mapserver -5.2.0 Beta4	mapserv.c 1334	*sprintf*	ABC: #include "MY_vsnprintf.h" if($MY\_vsnprintf$("%s%s%s.map",$mapserv$->map->$web.imagepath$,$mapserv$->map->$name$, $mapserv$->Id) >= $sizeof(buffer)$) return 1; AR: $snprintf(buffer, sizeof(buffer)$,"%s%s%s.map",$mapserv$ ->map->$web.imagepath$,$mapserv$->map->$name$, $mapserv$->Id);	AR: $snprintf(buffer, sizeof(buffer)$,"%s%s %s.$map$",$mapserv$->$map$->$web.$ $imagepath$,$mapserv$->map-> $name$,$mapserv$->Id);
cgminer -4.3.4	util.c 1883	*sprintf*	ABC: #include "MY_vsnprintf.h" if($MY\_vsnprintf$("%s:%s",$url, port$)>=$sizeof(address)$) return false; AR: $snprintf(address, sizeof(address)$,"%s:%s",$url, port$);	AR: $snprintf(address$,254,"%s:%s",url, $port$);

In the last two table rows, we used our self-defined function $MY\_vsnprintf()$ in header $MY\_vsnprintf.h$. It calls $vsnprintf$(NULL, 0, format, ...) to compute the length of the format string.

official developers. The repair of *man-1.5i2* using repair mode ABC and the repair of *mapserver=5.2.0Beta4* using repair mode AR of BovInspector are exactly the same as the official repairs. For *wwwcount-2.3*, in the AR repair mode, BovInspector repairs it by replacing *strcpy* with *strncpy* and setting the last element to '\0'. Official repair uses a $safeStrcpy$, which actually is a wrapper of our repair code. Therefore, they are essentially the same. The repair of *cgminer-4.3.4* using repair mode AR of BovInspector is also essentially the same as the official repair. The repair of *wu-ftpd-2.5.0* using repair mode AR is different from the official repair. In our repair, for the buffer *mapped_path* coming from the parameter, we can only find all its call sites and take the minimum value as the buffer size, which will be too conservative. The developer of *wu-ftpd* changed the definition of the function to make the size of the buffer available directly from a new parameter, i.e., *size* (but they wrongly set the last argument of *strncpy*, and reading the buffer may be out-of-bound.). For *gzip-1.2.4*, *sendmail-8.12.7* and *xmp-2.5.1*, BovInspector uses an `if(`$a >= b$`)` and the official repairs use an `if(`$a < b$`)` when adding boundary checks. It is up to the developers to choose which kind of boundary checks to be added in the programs.

9.4 Discussion

Experiments show that our method works well on buffer overflow detection and false alarm elimination.

Precision of Warning Validation: According to the warning classification rules shown in Subsection 3.2, for a warning, if we find there exists an overflowable path, then a test case that can trigger the buffer overflow will be generated, and thus the warning must be a true warning. If a warning is validated as a false warning, it indicates every reachable path for the warning is a safe path. Based on the warnings with suspicious buffer access information provided by static tools, we only check all the paths that cover these suspicious buffer operations. The manual inspection results have shown that all the validated false warnings by BovInspector are correct. But BovInspector can also conservatively ignore all the buffer operations in the warning (i.e., $\omega = (\langle\rangle, b)$). Then it will analyze all possible paths from the entry of a program to b. In that case, false warnings validated by BovInspector would be trustworthy. If there exists any path that exceeds the time limit and no overflowable path has been found, the warning will be classified as an undecided warning, which will need further manual inspection.

Our method can automatically validate about 60% of the buffer overflow warnings reported by Fortify for real-world programs, but there are still a lot of undecided ones. By analyzing the undecided cases, we find some main reasons.

Function Pointers: In the warning reachability analysis module, for function pointers, we simply skip the analysis inside the called functions. This will lead to the inaccuracy of ICFG in our guided symbolic execution module. As ICFG is used to determine which execution path cannot lead to the buffer overflow point, the execution path containing function pointers will not be removed when it is executed to the function pointers. In that case, all sub-paths of the function pointers will be explored during symbolic execution, which may lead to path explosion. Note that the inaccuracy of the ICFG may influence the effectiveness of our method; however, since our method considers all sub-paths in the function pointers, we can guarantee that the decisions are correct.

Loops and Branches: Another scenario is that some buffer overflow points are contained in a loop. During guided symbolic execution, our approach allows an execution to continue if the buffer overflow point is in a loop and the current overflowable path constraints cannot be solved. Then the executions can explore the rest of the loop. When an execution reaches the exit of the loop, there are two directions to be selected: one leads to the outer part of loop body, while the other jumps back to the loop entrance. In the warning reachability module, since the buffer overflow point is in the loop, the entrance of the loop is contained in the warning path. In that case, the execution will continue to explore the loop again, until the loop reaches the loop upper bound (which means the path constraints of the entrance branch cannot be solved) or the time upper bound, or it finds a solution to the overflowable path constraints at the buffer overflow point (which means it is a true warning). Some buffer overflows can only be triggered by a specific number of loop iterations, which will lead to the path explosion problem. Besides, the warning reachability module skips the analysis inside library functions, and without the guidance of warning reachability analysis, the symbolic execution engine will have to explore all the paths in the library functions if the library functions have been modeled by the symbolic execution engine. The number of library functions to be explored will grow exponentially with the number of branches, which will further increase the

number of branches. Moreover, the very existence of branches and loops will also increase the complexity of constraints. Therefore, sometimes, it may be impossible to solve constraints within a reasonable amount of time. In these situations, our method may not be able to decide whether corresponding warnings are true or false within the time limit and then an undecided warning will be reported.

Library calls: A warning point may not be reachable by the symbolic execution engine due to unknown library calls. When an execution path contains calls to the library functions that have been modeled by the symbolic execution engine, the engine will terminate the path that cannot be explored further. Besides, because there is a limitation in KLEE for accessing some parameters during symbolic execution or processing multi-threads and multi-processes systems, some buffer operations are not supported in BovInspector. In these situations, BovInspector will also treat corresponding warning as undecided one.

10 Related Work

Prior work on identifying buffer overflow vulnerabilities falls into two categories: static program analysis [3–7, 9] and dynamic execution analysis [13–17]. Besides, there are some researches on guided symbolic execution for test case generation [26–29] and automatic bug repair [30–33].

10.1 Static Program Analysis

Most static program analysis approaches scan software source code to discover the code segments that are possibly vulnerable to buffer overflow attacks. Each vulnerability warning needs to be manually inspected to check whether it is indeed a true vulnerability. ITS4 [3] scans C or C++ source code, breaks the codes into lexical tokens, and then matches patterns in the token stream to find possible vulnerabilities. Other similar tools include FlawFinder[13], which has more detailed report and supports more source code types. These schemes only consider the lexical information although they are simple and can be easily applied to large-scale programs.

Some schemes perform semantic analysis. BOON [4] focuses on string operations. By checking whether an operation can make the range of a variable outside the boundaries, BOON can report possible buffer overflow vulnerabilities. Splint [5] requires users to add source annotation to apply inter-procedural analysis and produces warnings for all library functions susceptible to buffer overflow vulnerabilities. These schemes share the same drawback of lacking run-time information so that they often report a large number of false alarms.

To improve accuracy, some schemes introduce path-sensitive analysis. ARCHER [6] adopts path-sensitive inter-procedural symbolic analysis on program source code, which reduces false alarms since some false positives are caused by infeasible paths. However, it cannot understand C string operations and spends too much resource for checking the paths not related to buffer overflow vulnerabilities. Marple [7] uses path-sensitive analysis to improve the detection accuracy, and classifies paths into five types: infeasible, safe, vulnerable, overflow-input-independence, and do not know. The main drawback of Marple is that the path-sensitive analysis is static, which means that it cannot identify the buffer overflows that need run-time information. Fabian *et al.* [9] designed a novel representation of source code called code property graph and with the comprehensive view of source code, they are able to model different common vulnerabilities more precisely. However, since this scheme does not interpret

[13] `https://dwheeler.com/flawfinder/`

code, it cannot find vulnerabilities induced by runtime behavior. AEG [8] mixes binary analysis with source code analysis to find exploitable vulnerabilities. It uses preconditioned symbolic execution to find bugs at the source code level. Then, it performs dynamic analysis in binary level with the input generated by symbolic execution to verify whether the vulnerability is exploitable. However, the precision is limited by the preconditions for symbolic execution.

There is also a large body of work focusing on symbolic execution or analysis on binary code or other languages. TaintScope [10] uses symbolic execution to automaticaly detect vulnerabilities via binary analysis. It performs a path sensitive static analysis to identify the vulnerable point on the intermediate representation, which is transformed from the disassembled code. Bitblaze [12] combines static, dynamic analysis, and program verification techniques to computer security. It can be used to detect buffer related security problems by extracting security-related properties from binary programs directly. Mergepoint [11] combines static symbolic execution and dynamic symbolic execution in binary code. It can be used for large-scale testing of commodity off-the-shelf (COTS) software.

Although many static analysis schemes for buffer overflow discovery have been proposed, it remains an open problem for scaling to large real systems and manually inspecting amounts of warnings. Our work is complementary to such schemes as our scheme can be used as a post-processing step for reducing the number of false positives.

10.2 Dynamic Program Analysis

The dynamic program analysis approach inserts special code into software so that buffer overflow occurrences can be detected and properly processed such as terminating software execution. Some dynamic analysis tools, such as StackGuard [13], add canaries before return addresses in the stack layout to protect entire distributions from stack smashing buffer overflow attacks. Some other methods [14–16] assume the boundary of variables should not be exceeded by all accesses and monitor the variables to find buffer overflow.

Splat [17] is a tool for automatically generating test cases for detecting buffer overflows and it performs directed random testing guided by symbolic execution. It uses symbolic length abstractions techniques to prune the state space without losing the buffer overflow detection ability. UndefinedBehaviorSanitizer (UBSan)[14] is a fast undefined behavior detector. It modifies the program at compile-time to catch various kinds of undefined behavior during program execution. All dynamic execution approaches of buffer overflows are challenged by generating high-quality test cases to trigger the bugs with limited efforts. Our approach is based on static analysis and symbolic execution, thus it is directed, automatic, and cost effective.

10.3 Guided Symbolic Execution

Guided symbolic execution techniques focus on controlling the procedure of symbolic execution and different methods are proposed to steer the exploration to various parts of the program to tackle the problem of path explosion. Generally, most guided symbolic execution techniques aim to improve the program coverage of the test suite generated. The control-flow guided search strategy [26] constructs a weighted control flow graph (CFG), guiding the exploration to the nearest currently uncovered parts based on the distance in the CFG when the concolic testing needs to choose branches to negate. The fitness-guided search strategy [29] calculates fitness values from explored paths to

[14] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

target predicates and fitness gains for the branches to be flipped, then selects proper paths and branches to cover the target predicates.

There are also some approaches which guide symbolic execution to specific parts of the program based on various purpose. The tool eXpress [27] introduces dynamic symbolic execution for regression test generation and prunes paths that do not expose behavioral differences while exploring new program versions. Domagoj *et al.* proposed a technique [28] which exploits static analysis to guide dynamic automated test generation for binary programs. This work is similar to ours, but it depends on the visibly pushdown automaton (VPA) generated by the seed tests, which may be not complete.

10.4 Automatic Bug Repair

Automatic bug repair is a promising way of reducing the cost, and many methods have been proposed recently. There are general techniques and fault-specific techniques existing for automatic bug repair [34, 35]. GenProg [30] uses genetic programming to guide the generate-and-validate process to repair defects in C programs. With new representation and mutation and crossover operators, GenProg can scale to large, open-source programs by taking advantages of cloud computing. RSRepair [31] presents a new automated repair technique using random search instead of genetic search. AE [32] also focuses on generate-and-validate repair methods. It uses a formal cost model which suggests an improved algorithm for defining and searching the space of patches and the order in which tests are considered. Qi *et al.* [33] presented a generate-and-validate patch generation system Kali which can achieve the same effect as prior works by just deleting functionality. CodePhage [36] automatically transfers correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. SearchRepair [37] exploits a database of human-written patches encoded as SMT formulas. These techniques try to automatically fix universal defects. Compared to these works, our method only repairs those validated true buffer overflow warnings, and is more targeted and simple. Currently, it is specific to buffer overflow vulnerabilities. By introducing more vulnerability models and investigating successful repair practices, our method can be extended for various defects. There are also some approaches that focus on repairing buffer overflow defects. DIRA [38] automatically instruments a network service program to detect control hijacking and record enough runtime information to generate the corresponding patch. It will extend a buffer according to its runtime information but the patch may be useless under another test case. TAP [39] is an automatic buffer overflow and integer overflow discovery and patching system. Its application is limited to those programs which contain incorrectly coded checks. ClearView [40] reallocates the compromised local array as a global array and sandwiches it in a pair of write-protected pages. But its patches are not similar to those that human programmers write. BovInspector generates automatic repair suggestions according to the 11 repair strategies preferred by programmers in Section 8 only for the true buffer overflow warnings validated in Section 7.

10.5 False Positives Elimination

There are several ways of eliminating false positives reported by static analysis. Ruthruff *et al.* [41] proposed logistic regression models to generate binary classifications of static analysis warnings. Their results show that the generated models were over 85% accurate in predicting false positives. However, this method cannot ensure the validity of the true warnings it predicted.

On the contrast, BovInspector will further provide a test case that can trigger the buffer overflow for each validated true warning.

The second way is using verification methods [42,43] such as model checking, symbolic execution to verify warnings reported by static analysis. Junker *et al.* [42] presented an abstraction refinement technique to automatically find and eliminate false positives. The analysis starts with a syntactic model according to the static program analysis. Then, it iteratively computes the infeasible sub-paths using SMT solvers and refines the model using additional automata. Muske *et al.* [43] proposed an observation-based method to avoid redundancy when using bounded model checking to verify false positives.

The last kind of methods leverages a precise static method to validate warnings reported by an imprecise but fast static method [44–46]. Smoke [44] shows a two-stage method that uses a precise static analysis method to eliminate false positives from the previous, imprecise but fast static method. Kim *et al.* [45] proposed to only perform a more precise analysis on the small fragments of the code that are more relevant to a buffer overflow alarm by invoking an SMT solver. In this way, they only try to remove false alarms as many as possible but cannot determine the truth of alarms. Finally, the most related work was proposed by Arzt *et al.* [46], which is a post-analysis step based on symbolic execution to prune infeasible paths from the result of data flow analysis. However, it is unsound since it only verifies one path among all possible paths for each warning. In contrast, our method verifies all possible paths that cover the buffer operations reported by a static tool to avoid false negatives. Moreover, our method also models buffer operations to not only verify whether a path is feasible, but also verify whether its overflowable constraints are satisfiable.

11 Conclusions

In this paper, we made three key contributions. First, we proposed the first framework called BovInspector for automatic validating buffer overflow warnings output by existing static program analysis tools. BovInspector is complementary to prior buffer overflow vulnerability discovery schemes, which are mostly based on static program analysis and sometimes symbolic execution. The key idea of BovInspector is to use symbolic execution to automatically identify those buffer overflow warnings that are true warnings or false warnings. The advantage of symbolic execution is its capability to explore program execution states that are unavailable to static program analysis. Second, we proposed the method for warning reachability analysis, guided symbolic execution, buffer overflow validation and targeted automatic repair. Third, we implemented BovInspector and evaluated its performance on both synthetic programs and real-world open source programs. The experimental results showed that BovInspector can significantly reduce the number of false alarms in buffer overflow vulnerability warnings outputted by static program analysis tools. In addition, with automatic repair, we provide suggestions to fix the true warnings or degrade them to dangerous functions.

In the future, we will extend our method to validate and repair other static software vulnerability warnings. Second, we will update the implementation of BovInspector with the latest LLVM[15] and KLEE[16]. Next, more accurate guidance of symbolic execution is a promising way to improve the efficiency of the validation. For example, many redundant and irrelevant

[15]http://llvm.org/

[16]https://klee.github.io/

paths can be pruned by more precise static analysis. In this case, guiding symbolic execution to only explore the core set of suspicious paths of a warning may significantly improve the performance of validation. Last, the scalability of our approach should be improved for industrial application.

References

[1] Anderson J P. Computer security technology planning study. volume 2. Technical report, Air Force Electronic Systems Division, Report ESD-TR-73-51, 1972.

[2] Shahzad M, Shafiq M Z, Liu A X. A large scale exploratory analysis of software vulnerability life cycles. In *Proc. of the 34th International Conference on Software Engineering*, June 2012.

[3] Viega J, Bloch J T, Kohno Y, McGraw G. Its4: A static vulnerability scanner for c and c++ code. In *Proc. of the 16th Annual Computer Security Applications Conference*, 2000, pp. 257–267.

[4] Wagner D A, Foster J S, Brewer E A, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of NDSS*, 2000, pp. 2000–02.

[5] Evans D, Larochelle D. Improving security using extensible lightweight static analysis. *IEEE Software*, 2002, 19(1):42–51.

[6] Xie Y, Chou A, Engler D. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003, pp. 327–336.

[7] Le W, Soffa M L. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 272–282.

[8] Avgerinos T, Cha S, Hao B, Brumley D. Aeg: Automatic exploit generation. In *Proc. of NDSS*, volume 11, 2011, pp. 59–66.

[9] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of S&P*, 2014, pp. 590–604.

[10] Wang T, Wei T, Gu G, Zou W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proc. of S&P*, 2010, pp. 497–512.

[11] Avgerinos T, Rebert A, Cha S, Brumley D. Enhancing symbolic execution with veritesting. In *Proc. of the 36th International Conference on Software Engineering*, 2014, pp. 1083–1094.

[12] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang M G, Liang Z, Newsome J, Poosankam P, Saxena P. Bitblaze: A new approach to computer security via binary analysis. In *Proc. of the International Conference on Information Systems Security*, pp. 1–25. Springer, 2008.

[13] Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium*, volume 98, 1998, pp. 63–78.

[14] Jones R W, Kelly P H. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Inte. Workshop on Automated Debugging*, 1997, pp. 13–26.

[15] Wagner D, Dean R. Intrusion detection via static analysis. In *Proc. of the IEEE Symposium on Security and Privacy*, 2001, pp. 156–168.

[16] Haugh E, Bishop M. Testing c programs for buffer overflow vulnerabilities. In *Proc. of NDSS*, 2003.

[17] Xu R G, Godefroid P, Majumdar R. Testing for buffer overflows with length abstraction. In *Proc. of the international symposium on Software testing and analysis*, 2008, pp. 27–38.

[18] Gao F, Wang L, Li X. Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities. In *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 786–791.

[19] Clarke L A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 1976, SE-2(3):215–222.

[20] Cadar C, Dunbar D, Engler D R et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of OSDI*, volume 8, 2008, pp. 209–224.

[21] Ye T, Zhang L, Wang L, Li X. An empirical study on detecting and fixing buffer overflow bugs. In *Proc. of the IEEE International Conference on Software Testing, Verification and Validation*, 2016, pp. 91–101.

[22] Yu F, Bultan T, Ibarra O H. Symbolic string verification: Combining string analysis and size analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 322–336.

[23] Larochelle D, Evans D. Statically detecting likely buffer overflow vulnerabilities. In *Proc. of the 10th USENIX Security Symposium*, volume 32, 2001, pp. 177–190.

[24] Zitser M, Lippmann R, Leek T. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 97–106.

[25] Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[26] Burnim J, Sen K. Heuristics for scalable dynamic test generation. In *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[27] Taneja K, Xie T, Tillmann N, De Halleux J. eXpress: guided path exploration for efficient regression test generation. In *Proc. of the International Symposium on Software Testing and Analysis*, 2011, pp. 1–11.

[28] Babić D, Martignoni L, McCamant S, Song D. Statically-directed dynamic automated test generation. In *Proc. of the International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[29] Xie T, Tillmann N, De Halleux J, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. of the IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 359–368.

[30] Le Goues C, Dewey-Vogt M, Forrest S, Weimer W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. of the 34th International Conference on Software Engineering*, 2012, pp. 3–13.

[31] Qi Y, Mao X, Lei Y, Dai Z, Wang C. The strength of random search on automated program repair. In *Proc. of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[32] Weimer W, Fry Z P, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proc. of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 356–366.

[33] Qi Z, Long F, Achour S, Rinard M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. of the International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[34] Gazzola L, Micucci D, Mariani L. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 2017, 45(1):34–67.

[35] Monperrus M. Automatic software repair: a bibliography. *ACM Computing Surveys*, 2018, 51(1):17.

[36] Sidiroglou-Douskos S, Lahtinen E, Long F, Rinard M. Automatic error elimination by horizontal code transfer across multiple applications. In *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 43–54.

[37] Ke Y, Stolee K T, Le Goues C, Brun Y. Repairing programs with semantic code search (t). In *Proc. of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 295–306.

[38] Smirnov A, Chiueh T c. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Proc. of NDSS*, 2005.

[39] Sidiroglou-Douskos S, Lahtinen E, Rinard M. Automatic discovery and patching of buffer and integer overflow errors. *In CSAIL Technical Reports*, 2015.

[40] Perkins J H, Kim S, Larsen S, Amarasinghe S, Bachrach J, Carbin M, Pacheco C, Sherwood F, Sidiroglou S, Sullivan G et al. Automatically patching errors in deployed software. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 87–102.

[41] Ruthruff J, Penix J, Morgenthaler J, Elbaum S, Rothermel G. Predicting accurate and actionable static analysis warnings. In *Proc. of the ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 341–350.

[42] Junker M, Huuck R, Fehnker A, Knapp A. Smt-based false positive elimination in static program analysis. In *International Conference on Formal Engineering Methods*, 2012, pp. 316–331.

[43] Muske T, Datar A, Khanzode M, Madhukar K. Efficient elimination of false positives using bounded model checking. In *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*, 2013, pp. 13–20.

[44] Fan G, Wu R, Shi Q, Xiao X, Zhou J, Zhang C. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *Proc. of the IEEE/ACM 41st International Conference on Software Engineering*, 2019, pp. 72–82.

[45] Kim Y, Lee J, Han H, Choe K M. Filtering false alarms of buffer overflow analysis using smt solvers. *Information and Software Technology*, 2010, 52(2):210–219.

[46] Arzt S, Rasthofer S, Hahn R, Bodden E. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, 2015, pp. 1–6.

Feng-Juan Gao is a Ph.D. candidate in Nanjing University, Nanjing. She received her B.S. degree in computer science from University of Electronic Science and Technology of China, Chengdu, in 2014. Her research is in software engineering, with focus on symbolic execution.

Yu Wang is a Ph.D. candidate in Nanjing University, Nanjing. He received his B.S. degree in computer science from University of Electronic Science and Technology of China, Chengdu, in 2014. His research is in software engineering, with focus on analyzing concurrent software defects.

Lin-Zhang Wang is a professor in State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2005. His research interests include software engineering and software testing.

Zijiang Yang is a Professor at Western Michigan University. He received his Ph.D. degree in computer science from the University of Pennsylvania, Philadelphia, in 2003. His research interests are in the broad areas of software engineering and formal methods.

Xuan-Dong Li is a professor in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his B.S. degree in 1985, M.S. degree in 1991, and Ph.D. degree in 1994, all in computer science from Nanjing University, Nanjing. His research interests include software modeling and analysis, software testing and verification.