BovInspector: Automatic Inspection and Repair of Buffer Overflow Vulnerabilities

Fengjuan Gao
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
fjgao@seg.nju.edu.cn

Linzhang Wang
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
lzwang@nju.edu.cn

Xuandong Li
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
lxd@nju.edu.cn

ABSTRACT

Buffer overflow is one of the most common types of software vulnerabilities. Various static analysis and dynamic testing techniques have been proposed to detect buffer overflow vulnerabilities. With automatic tool support, static buffer overflow detection technique has been widely used in academia and industry. However, it tends to report too many false positives fundamentally due to the lack of software execution information. Currently, static warnings can only be validated by manual inspection, which significantly limits the practicality of the static analysis. In this paper, we present BovInspector, a tool framework for automatic static buffer overflow warnings inspection and validated bugs repair. Given the program source code and static buffer overflow vulnerability 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 fix it with three predefined strategies. BovInspector is complementary to prior static buffer overflow discovery schemes. Experimental results on real open source programs show that BovInspector can automatically inspect on average of 74.9% of total warnings, and false warnings account for about 25% to 100% (on average of 59.9%) of the total inspected warnings. In addition, the automatically generated patches fix all target vulnerabilities. Further information regarding the implementation and experimental results of BovInspector is available at http://bovinspectortool.github.io/project/. And a short video for demonstrating the capabilities of BovInspector is now available at https://youtu.be/IMdcksROJDg.

CCS Concepts

•Software and its engineering → Software verification and validation; Software defect analysis; Software testing and debugging; •Security and privacy → Vulnerability scanners;

Keywords

Buffer Overflow; Symbolic Execution; Validation; Automatic Repair;

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970282

1. INTRODUCTION

Buffer overflow occurs when a program writes data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. The buffer overflow vulnerability provides a way for attackers to corrupt data, crash the program, or execute malicious code. Attackers often carefully construct their inputs to overflow the buffer and overwrite the adjacent memory which may contain return address to execute their malicious code. Although buffer overflow has been explored for many years, it remains one of the most common types of software vulnerabilities according to the statistics by Common Vulnerabilities and Exposures (CVE) [1]. Buffer overflow attacks against both legacy and newly-deployed software systems can lead to system crash, denial of service, or loss of control to external attackers, leading to disastrous consequences.

Challenge. Static program analysis and dynamic testing approaches have been proposed to detect buffer overflow vulnerabilities. Dynamic testing methods are commonly used during software deployment [12][20][15], however, they highly rely on the completeness of the test suite. Static analysis approaches can achieve high level automation, which makes them popular in practice, such as [7][11][13][19]. However, the key limitation of static technique is that the reported buffer overflow vulnerabilities warnings contain too many false positives fundamentally due to the lack of software execution information. Each static vulnerability warning needs to be manually inspected to identify true vulnerabilities and false alarms. Our experience shows that even commercial (and mature) static analysis tools like HP Fortify [7], which is one of the leading commercial products in application security market[1], can report a great number of warnings for a moderate-sized program. As shown in Table 1, as the scale of program increases, the total number of warnings will also increase notably. It's challenging for programmers to manually inspect all of the warnings and source code to

[1]Gartner Magic Quadrant for Static Application Security Testing, https://www.gartner.com/doc/2538715/magic-quadrant-application-security-testing

Table 1: Warnings Reported by Fortify on Real World Programs

Program	Scale	Total Number of Warnings	Total Number of Buffer Overflow Warnings
gzip1.2.4	5.1K	237	19
tftp-hpa5.0	5.3K	78	12
net-tool-1.60	8.1K	1256	62
inspircd2.0.5	74.1K	142	11
udisks2.1.2	92.1K	437	8
tiff4.0.3	113.4K	1743	1117
freetype2.4.8	205.6K	479	16
firefox-33.0	683.3K	23990	4204

find and repair true vulnerabilities, which is a daunting task that can be extremely tedious, labor-intensive, and time-consuming. In addition, the false positives waste a huge amount of human effort on manual inspection.

Our Proposal. In this paper, we present an automatic framework, BovInspector, to help programmers to inspect buffer overflow vulnerability warnings output by existing static program analysis tools for C programs, as well as repair validated true buffer overflow vulnerabilities. First, BovInspector performs warning reachability analysis to find reachable path segments of each static buffer overflow warning in the control flow graph of the program. Next, BovInspector explores all the program paths guided by previous generated reachable path segments of each reachable warning in an extended symbolic execution engine. Then, each reachable warning is validated and classified by checking whether all the path conditions and the buffer overflow constraints can be satisfied simultaneously during the symbolic execution. Finally, for each validated true warning, BovInspector fixes it by adding boundary checks or replacing unsafe APIs with safe ones.

So far, BovInspector is complementary to prior static buffer overflow detection techniques. We make the following contributions:

- We combine static analysis and dynamic symbolic execution technique to automatically inspect buffer overflow warnings, which will then be classified into true warnings, false warnings, and undecidable warnings.
- We provide three strategies to automatically repair validated true buffer overflow vulnerabilities, i.e., adding boundary checks, replacing the API with a safer one, extending buffers, according to empirical study of human repair of buffer overflow vulnerabilities in real world projects.
- We implement a prototype tool BovInspector to support for automatically inspecting warnings and repairing vulnerabilities.

2. METHOD

Figure 1 shows the overall framework of BovInspector, which mainly consists of four steps: warning reachability analysis, guided symbolic execution, buffer overflow validation, and targeted automatic repair. The key idea of BovInspector is to use symbolic execution to automatically identify those buffer overflow warnings reported by static analysis that are true warnings or false warnings. To avoid the infamous path explosion issue of symbolic execution, BovInspector uses static analysis to guide the symbolic execution so that it only focuses on the execution paths that cover the buffer overflow warnings generated by static program analysis. After validating the buffer overflow vulnerabilities, BovInspector will automatically repair these buffer overflow vulnerabilities according to popular human repair strategies.

2.1 Warning Reachability Analysis

Given the source code of a program, we first use Fortify to perform a static analysis to get all the buffer overflow warnings. Each buffer overflow warning can be represented as a tuple: (d,<l_1,l_2,...,l_n>,o), in which "d" is the statement label where the buffer is declared, "$l_i(1 \le i \le n)$" is the statement label where an operation is performed on the buffer, and "o" is the statement label where the buffer overflow may occur. The statement label including file name and line number represents where the statement is. To confirm whether the buffer overflow warning point is reachable, we first build CFG for each procedure from the source code. Each basic block in CFG will be packaged into a *CFGNode* to record its predecessor and successor. Then, we generate the Interprocedural Control Flow Graph (ICFG) according to the calling relationships among procedures. Next, we try to obtain the reachability of the warning points on the ICFG. If we just perform a depth-first traversal on the ICFG, we will get many paths that does not contain any path segments in the buffer overflow warnings. So based on our bi-direction ICFG, for each warning point, we perform backward tracking on the ICFG starting from the warning point, which can significantly reduce the number of useless paths. For a warning point, a warning path set is a set of complete paths covering all the reachable path segments of the warning, starting from the program entrance and ending at the warning point. And a warning point may have more than one warning path set. If there is such a warning path set existing for a warning point, we consider the warning point is reachable. Otherwise, the warning point is unreachable, and the corresponding warning point will be classified as a false warning.

2.2 Guided Symbolic Execution

Symbolic execution is commonly used in software testing, which executes a program with symbolic inputs. In this paper, we use the KLEE [9] symbolic execution engine. But traditional symbolic execution has an infamous path explosion issue. To avoid path explosion, BovInspector uses *Warning Reachability analysis*'s results to guide KLEE's symbolic execution to only focus on the warning paths. And that means we should remove the execution states not corresponding to any of the warning paths. The symbolic execution engine maintains an execution state pool containing all the executed states. And the engine fetches program counter from the pool and interprets the instruction pointed by the counter. If the instruction is a branch command, the symbolic execution engine will duplicate the current execution state, set the two execution states' next instruction to the true direction and false direction of the branch instruction separately and add the path constraint to the corresponding execution state. Next, we need to decide which direction of the branch instruction to choose to reach the warning point, namely which execution state to choose during symbolic execution. Each instruction can be represented by its basic block's entrance statement label. If both states' statement labels are in the warning path set, we will not remove any of the state. If only one state's statement label is in the warning path set, then we will remove the other state from the state pool. If both states' statement labels are not in the warning path set,

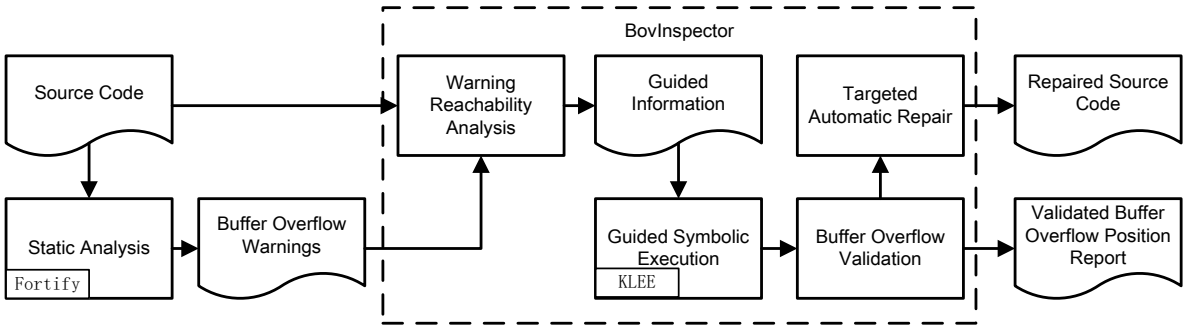

Figure 1: The Framework of BovInspector

we will not remove any of the states because sometimes the symbolic execution engine will explore inside some library calls which are not in the warning path set and sometimes the buffer overflow will be triggered by multiple execution of a loop. Each time when an instruction is interpreted, the symbolic execution engine will select an execution state from the state pool to perform the next execution, which will continue until there's no state in the execution state pool or the time threshold value is exceeded.

2.3 Buffer Overflow Validation

KLEE can only identify some common defects such as null pointers and out-of-bound pointers during runtime. To validate buffer overflow warnings during symbolic execution, we extend KLEE's symbolic execution engine by adding a buffer overflow validation module. During the guided symbolic execution, the extended executor will collect the path conditions. When encountering a buffer overflow warning point, the extended executor will duplicate the current execution state and add buffer overflow constraints (what buffer overflow constraints need to be added will be discussed in the following contents) into its corresponding backup state. Then the constraint solver will be invoked to determine whether the path conditions together with the buffer overflow constraints are satisfiable. Apparently, there may be more than one reachable path for a buffer overflow warning point. If one of the reachable path's constraints together with the buffer overflow constraints are satisfiable, then the buffer overflow warning reported by Fortify is a true warning and the specific method will be invoked to generate a test case that will follow the path and trigger the buffer overflow. If none of the reachable path's constraints together with the buffer overflow constraints are satisfiable, then the buffer overflow warning reported by Fortify is a false warning. If the constraints cannot be solved in the given time threshold, then the buffer overflow warning reported by Fortify is an undecidable warning.

First, to determine which APIs should be focused on in our BovInspector, we studied APIs operating buffers in C99 [8] and Linux system call interface. The APIs we focus on in BovInspector are copying functions, concatenation functions, format input/output functions, character input/output functions and file input/output functions. In addition to these APIs, we also care about direct buffer accessing, namely array and pointer accessing.

Next, we need to determine what condition should be satisfied to trigger a buffer overflow vulnerability. As a buffer overflow occurs when more data is written into a buffer than it can hold, we need to figure out the targeted buffer's real capacity and the original data's possible length. To trigger a buffer overflow, the original data's possible length must be bigger than the targeted buffer's real capacity, which constructs the buffer overflow constraints. Accordingly, to avoid a buffer overflow, the boundary check need to be added will be that if buffer overflow constraints are satisfiable, the program must be terminated.

Finally, based on the above analysis, we propose our "Buffer Overflow Constraints and Repair Models", which is shown in Table 2. Particularly, we put direct buffer accessing in the model. Direct buffer accessing includes array and pointer accessing. Column 2 shows the APIs that BovInspector can handle with, column 3 shows what parameters format the APIs use, column 4 shows what constraint must be satisfied to trigger a buffer overflow, and column 5 shows what API can be used to replace the corresponding APIs to secure the program. In column 4, "strlen" and "sizeof" are C language library functions. "strlen" returns the length of the C string without including the terminating null character [2]. And "sizeof" returns the size of data type or variables [3]. Take "strcpy" for example to explain how we construct the "Buffer Overflow Constraints", "strcpy" copies the C string pointed by "src" into the array pointed by "dest", including the terminating null character (and stopping at that point) [4]. To trigger a buffer overflow, "strlen(src)+1" ("+1" is because the source string includes a terminating null character '\0') should be bigger than "sizeof(dest)", which means "Buffer Overflow Constraints" of "strcpy" will be "strlen(src) $\geq$ sizof(dest)", and so on.

2.4 Targeted Automatic Repair

Tao et al. [21] perform an empirical study on detecting and fixing buffer overflow bugs. More specifically they investigate officially adopted or programmers' preferred fix approaches for buffer overflow vulnerabilities in real world programs reported in CVE [1]. The investigation shows that adding boundary checks turns out the most common way (nearly 48%) to fix buffer overflow vulnerabilities, followed by replacing API with a safer API and extending the buffer size. BovInspector implements the top three of the most popular buffer overflow repair strategies according to this conclusion. Currently, BovInspector can handle buffer overflow vulnerabilities caused by the APIs listed in Table 2.

We design three repair strategies for BovInspector, i.e., "default" for adding boundary checks, "API-REP" for replac-

Table 2: Buffer Overflow Constraints and Repair Models

Type	API	Parameters Format	Buffer Overflow Constraints	Safer API Option
1	strcpy	(char *dest, const char *src)	strlen(src) ≥ sizeof(dest)	strncpy(dest,src,sizeof(dest))
2	strncpy memcpy memmove memset snprintf vsnprintf	(char *dest, const char *src, size_t n)	n > sizeof(dest)	
3	strcat	(char *dest, const char *src)	strlen(src)+strlen(dest) ≥ sizeof(dest)	snprintf(dest+strlen(dest), sizeof(dest)-strlen(dest),"%s",src)
4	strncat	(char *dest, const char *src, size_t n)	min{strlen(src),n}+strlen(dest) ≥ sizeof(dest)	
5	sprintf	(char *str, const char *format, ...)	format string length ≥ sizeof(str)	snprintf(str,sizeof(str),format,...)
6	fgets	(char *str, int num, FILE *stream)	num > sizeof(str)	
7	fread	(void *ptr, size_t size, size_t count, FILE *stream)	size*count > sizeof(ptr)	
8	read	(int fd, void *buf, size_t count)	count > sizeof(buf)	
9	buf[i], (buf+i)		i*typesize ≥ sizeof(buf)	

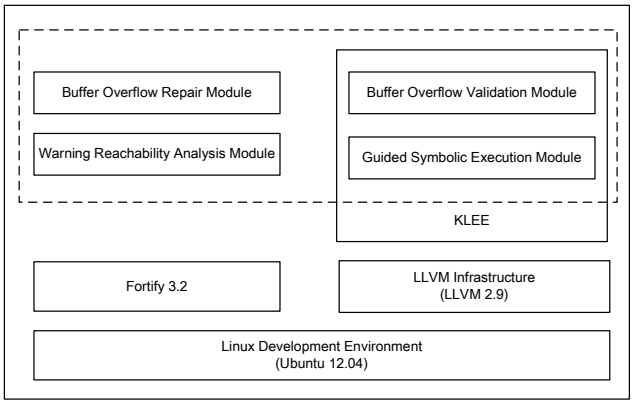

Figure 2: BovInspector Tool Architecture

ing the API with a safer API and "extend" for modifying the buffer's definition to extend its buffer. BovInspector takes source code files that need to be repaired and their buffer overflow vulnerabilities reports as inputs, and outputs repaired source code files.

BovInspector reads the buffer overflow vulnerabilities reports and extracts the position of each buffer overflow vulnerability and the targeted API. The reports may contain buffer overflow vulnerabilities at multiple locations in different source files. If we insert some statements in the source code to repair the first vulnerability, it will result in a line number mismatch between the source code and the report for the other vulnerabilities. So BovInspector first extracts all the buffer overflow vulnerabilities reports in each source file and then sorts the positions of buffer overflow vulnerabilities in each source file by the line numbers in descending order. Then BovInspector repairs the buffer overflow vulnerabilities in each source file according to the above order. For each buffer overflow vulnerability, BovInspector first locates the corresponding source code and fetches the actual parameters from it. And the required parameters are listed in the fourth column of Table 2. Given the targeted API, required parameters and repair mode, BovInspector will return the corresponding repair suggestions:

(1) Adding boundary checks: In this mode, BovInspector will add a boundary check in the previous line of the buffer overflow line. And the boundary check is an "if" statement with the condition constructed from the fourth column of Table 2 with the actual parameters. Specifically, if the targeted API is "sprintf", we will compute the length of format string by setting the first parameter of "vsnprintf" [5] to "NULL" and the second to "0".

(2) Replacing the API with a safer API: In this mode, BovInspector will rewrite the buffer overflow line with a safer API calling constructed from the fifth column of Table 2 with the actual parameters. Specifically, as Table 2 shows, there are only three types of APIs having a "Safer API Option". The other types of APIs will be repaired by adding boundary checks in this mode.

(3) Extending buffers: In this mode, BovInspector will need the definition position of the buffer that may overflow, which can be fetched from the *Buffer Overflow Warnings*. But the buffer size is always program specific and we will let the programmers configure the buffer size.

3. IMPLEMENTATION AND EVALUATION

BovInspector is built on the LLVM-2.9 compiler infrastructure [10]. It is composed of four modules, warning reachability analysis, guided symbolic execution, buffer overflow validation, and buffer overflow repair, as shown in Figure 2. Warning reachability analysis module is implemented as a LLVM pass [6] to build control flow graph (CFG) of the program, analyze the reachability of buffer overflow warnings in the graph, and finally output warning path sets. We extend the function of KLEE [9], a state-of-the-art symbolic execution engine, to support guided symbolic execution module and buffer overflow validation module. Buffer overflow repair module is implemented as a python script with three repair strategies. The static buffer overflow detector used in our experiments is HP Fortify with version 3.2 [7]. We provide interface to process warnings reported by Fortify. It is very straight forward to work with other static analysis tools.

To evaluate the capability of BovInspector, we have conducted control experiments on a benchmark. To prepare the benchmark, we select 8 programs from GNU COREUTILS utilities and real-world open source programs such as sendmail-8.12.7, gzip-1.2.4, and so on. And the programs

Table 3: Inspection and Repair Results on Real Program

Program	BovPosition	BovCause	Fortify(before)	BovInspector	Fortify(after)	
					Bound Checking	API-Rep
gzip-1.2.4	gzip.c:1009	strcpy	Bov	Bov	Dangerous function	Nothing
wwwcount-2.3	main.c:346	strcpy	Bov	Bov	Dangerous function	Nothing
net-tool-1.6	netstat.c:450,457, 602,608,737,743	strcat	Bov	Bov	Nothing	Nothing

Table 4: Results of the BovInspector's Repair and the Official Repair

Program	BovPosition	BovCause	BovInspector Repair	Official Repair
gzip-1.2.4	gzip.c:1009	strcpy	①if(strlen(iname)>= sizeof(ifname)) ②strncpy(ifname,iname,sizeof(ifname));	if(sizeof ifname - 1<= strlen (iname))
man-1.5i2	man.c:299	strcpy	①if(strlen(name0)>= sizeof(ultname)) ②strncpy(ultname,name0,sizeof(ultname));	if(strlen(name0) >= sizeof(ultname))
wu-ftpd-2.5.0	ftpd.c:1210	strcpy	①if(strlen(mapped_path) >= sizeof(path)) ②strncpy(path,mapped_path,sizeof(path));	strncpy(path, mapped_path, size);
xmp-2.5.1	dtt_load.c:79	array	①if(i*sizeof(pofs[0]) >= sizeof(pofs))	if (i < 256)
mapserver-5.2.0 Beta4	mapserv.c: 1334	sprintf	①#include "MY_vsnprintf.h" if(MY_vsnprintf("%s%s%s.map",mapserv ->map->web.imagepath,mapserv->map ->name,mapserv->Id) >= sizeof(buffer)) ②snprintf(buffer,sizeof(buffer),"%s%s%s. map",mapserv->map->web.imagepath, mapserv->map->name,mapserv->Id);	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	①#include "MY_vsnprintf.h" if(MY_vsnprintf("%s:%s",url,port)>= sizeof(address)) ②snprintf(address,sizeof(address),"%s:%s", url,port);	snprintf(address,254,"%s:%s",url,port);

are compiled into LLVM bitcode by LLVM-gcc. All experiments are conducted on a quad-core machine with an Intel Core (TM) Corei5-2400 3.10GHz processor and 4G memory, running Linux 3.11.0. Due to the page limit, all the subjects and evaluation results of the automatic inspection are put online at http://bovinspectortool.github.io/project/.

The results of automatic repair are shown in Table 3. The given "BovPosition" in Table 3 are all real buffer overflow vulnerabilities that have been validated officially. The "BovInspector" column shows that our BovInspector validates all these buffer overflow warnings reported by Fortify as buffer overflow vulnerabilities. After all the real vulnerabilities are repaired by BovInspector, the program will be rescanned by Fortify to verify BovInspector's repair ability. And the results show that Fortify will report no warnings on the programs repaired by BovInspector by adding boundary checks or replacing with a safer API. And BovInspector's repair results are also validated by having a human review. So our BovInspector can automatically and correctly repair the buffer overflow vulnerabilities that have been validated by itself.

To further illustrate the correctness of BovInspector's repair, we select 6 real world programs, and compare the patches generated by BovInspector and the official manual repair. And we obtain both the buggy and fixed versions of programs that contain buffer overflow vulnerabilities. By comparing the buggy and fixed versions of a program, we will know where the buffer overflow vulnerabilities are. We add the above buffer overflow vulnerabilities' positions into the buffer overflow vulnerabilities reports which will be used to guide the BovInspector's repair. After using BovInspector to repair the buffer overflow vulnerabilities in the source code, we will use Fortify to perform a static analysis on the repaired source code. Meanwhile, we analyze where and how the programmers repaired the buffer overflow vulnerabilities. Next, we compare with the differences between the BovInspector's repair and the official repair. Table 4 shows the results of the BovInspector's repair and the official repair. The "BovInspector Repair" column shows the results of two ways to repair buffer overflow vulnerabilities. The first method is adding boundary checks and the second method is replacing the buggy line with a safer API call. We list both the two repair methods' results to further show how BovInspector repairs buffer overflow vulnerabilities. The results show that BovInspector's repair methods are very similar to the official repair methods, namely similar to the human programmers' repair habit.

4. RELATED WORK

Static analysis approaches such as [11][13][19] scan software source code to discover possible buffer overflows. Static analysis methods can achieve high level automation, which makes them popular in commercial tools such as [7]. These static analysis tools are easy to use, but they often report a large number of false warnings due to the lack of runtime information and adopted conservative strategies in static techniques. In BovInspector, the static analysis tool Fortify [7] is used as a pre-processing tool to guide BovInspector's symbolic execution to only focus on the warning paths, which will raise the efficiency of symbolic execution and buffer overflow validation. And BovInspector can inspect the Fortify' buffer overflow warnings as true warnings or false warnings to reduce its large number of false positives. Namely, our method is complementary to the static analysis methods.

Dynamic testing methods are commonly used during software deployment. Hossain et al. [15] propose a mutation-based testing technique to generate adequate test data set for buffer overflow vulnerabilities. Eric et al. [12] develop a tool that instruments programs to keep track of memory

buffers and checks arguments to functions to determine if they satisfy certain conditions and warns when a buffer overflow may occur. The testing completeness of these methods highly relies on the completeness of the test suite, which cannot be guaranteed. And without a target guidance, these testing methods always generate a lot of test cases that cannot trigger a real vulnerability. In BovInspector, we use symbolic execution to automatically generate test cases that achieve high coverage on complex programs. What's more, we use guided symbolic execution to only focus on the buffer overflow warning paths reported by Fortify to reduce the useless path execution and test case generation.

As high level automation has become a trend, many methods and tools are proposed to perform automatic bug fix. DIRA [18] automatically instruments a network service program to detect control hijacking and record enough runtime information to generate the corresponding patch. The patches are similar to our BovInspector except that DIRA will extend a buffer according its runtime information but the patch may be useless under another test case. And DIRA detects control hijacking at runtime which depends on test cases to find bugs. BovInspector doesn't need the runtime information and it can inspect and repair buffer overflow vulnerabilities during software development. TAP [17] is an automatic buffer and integer overflow discovery and patching system. Its application range is limited to those programs which contain incorrectly coded checks. What's more, it can only insert boundary checks to repair these buffer overflows. Our BovInspector's buffer overflow detection module does not require the program to contain incorrectly coded checks. And BovInspector provides more ways to repair buffer overflow vulnerabilities for the programmers. ClearView [14] 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. And our BovInspector's repair module is designed and implemented according to programmers suggested repair methods. CodePhage [16] automatically transfers correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. CodePhage is different from BovInspector's repair method, because it relies on the existence of a specific donor application containing of the exact program logic to fix an error.

5. CONCLUSIONS

In this paper, we propose an automatic inspection and repair tool BovInspector for C programs. For each static buffer overflow warning, BovInspector first performs warning reachability analysis, and identifies reachable path segments for the reachable warning. Then, it uses guided symbolic execution to explore all the reachable path segments regarding to the warning. Based on the buffer overflow model and runtime information collected during the symbolic execution, the warning can be validated and classified as true warning, false warning, or undecidable warning. Finally, each true buffer overflow vulnerability could be automatically repaired with three strategies. The experimental results show that BovInspector can automatically and correctly inspect static buffer overflow warnings, and repair validated buffer overflow vulnerabilities.

6. ACKNOWLEDGMENTS

The paper was partially supported by the National Grand Fundamental Research 973 Program of China (No.2014CB34 0703) and the National Natural Science Foundation of China (No. 91418204, 61321491, 61472179, 61561146394, 61572249). We thank all the students who helped us in the experiment.

7. REFERENCES

[1] http://www.cvedetails.com/index.php.
[2] http://www.cplusplus.com/reference/cstring/strlen/.
[3] http://en.cppreference.com/w/cpp/language/sizeof.
[4] http://www.cplusplus.com/reference/cstring/strcpy/.
[5] http://www.cplusplus.com/reference/cstdio/vsnprintf/.
[6] http://llvm.org/docs/WritingAnLLVMPass.html.
[7] Hp fortify static code analyzer. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.
[8] Iso/iec 9899:1999. http://www.iso.org/iso/catalogue_detail.htm?csnumber=29237.
[9] Klee llvm execution engine. https://klee.github.io/.
[10] The llvm compiler infrastructure. http://llvm.org/.
[11] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE software'02*, 19(1):42–51.
[12] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities. In *NDSS'03*.
[13] W. Le and M. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *FSE'08*, pages 272–282.
[14] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, et al. Automatically patching errors in deployed software. In *SOSP'09*, pages 87–102.
[15] H. Shahriar and M. Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *COMPSAC'08*, pages 979–984.
[16] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *PLDI'15*, pages 43–54.
[17] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard. Automatic discovery and patching of buffer and integer overflow errors. *CSAIL Technical Reports'15*.
[18] A. Smirnov and T. Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS'05*.
[19] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS'00*, pages 2000–02.
[20] R. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA'08*, pages 27–38.
[21] T. Ye, L. Zhang, L. Wang, and X. Li. An empirical study on detecting and fixing buffer overflow bugs. In *ICST'16*, pages 91–101.