2016 IEEE International Conference on Software Testing, Verification and Validation

An Empirical Study on Detecting and Fixing Buffer Overflow Bugs

Tao Ye*, Lingming Zhang†, Linzhang Wang*, Xuandong Li*
*State Key Laboratory of Novel Software Technology, Nanjing University, 210023, China
*Jiangsu Novel Software Technology and Industrialization, Nanjing 210023, China
yt@seg.nju.edu.cn, lzwang@nju.edu.cn, lxd@nju.edu.cn
†Department of Computer Science, University of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

***Abstract*—Buffer overflow is one of the most common types of software security vulnerabilities. Although researchers have proposed various static and dynamic techniques for buffer overflow detection, buffer overflow attacks against both legacy and newly-deployed software systems are still quite prevalent. Compared with dynamic detection techniques, static techniques are more systematic and scalable. However, there are few studies on the effectiveness of state-of-the-art static buffer overflow detection techniques. In this paper, we perform an in-depth quantitative and qualitative study on static buffer overflow detection. More specifically, we obtain both the buggy and fixed versions of 100 buffer overflow bugs from 63 real-world projects totalling 28 MLoC (Millions of Lines of Code) based on the reports in Common Vulnerabilities and Exposures (CVE). Then, quantitatively, we apply `Fortify`, `Checkmarx`, and `Splint` to all the buggy versions to investigate their false negatives, and also apply them to all the fixed versions to investigate their false positives. We also qualitatively investigate the causes for the false-negatives and false-positives of studied techniques to guide the design and implementation of more advanced buffer overflow detection techniques. Finally, we also categorized the patterns of manual buffer overflow repair actions to guide automated repair techniques for buffer overflow. The experiment data is available at http://bo-study.github.io/Buffer-Overflow-Cases/.**

I. Introduction

In an unmanaged language such as C/C++, programmers need to explicitly and manually deal with memory manipulations. Inappropriate memory manipulation, mistaken assumptions about the size or makeup of a piece of data, and misuse of API may result in violation of a programmer's assumption during the runtime, and can easily lead to security vulnerabilities. Buffer overflow is one of the security vulnerabilities that are caused by missing input validation or bounds checking before memory manipulation or API calling, and can easily overwrite the allocated bounds of the buffers they operate upon.

Buffer overflow has become one of the best known types of software security vulnerabilities. Although researchers have proposed various techniques for buffer overflow detection, buffer overflow attacks against both legacy and newly-deployed software systems are still quite prevalent. According to the statistics by Common Vulnerabilities and Exposures (CVE) [6], buffer overflow accounts for 14.6% of all software vulnerabilities since 1999. With Code Execution for 31.4% and Denial of Service for 21.5%, buffer overflow is now the third most popular type of vulnerabilities. Software containing buffer overflow bugs can cause system crash, denial of service, or loss of control to external attackers, leading to disastrous consequences.

Currently, there are two general approaches to identifying buffer overflow vulnerabilities: static program analysis [1], [2], [8], [18], [20], [21], [23]–[27], [29], [31], [32] and dynamic testing [5], [12], [14], [19], [30]. The dynamic testing approach inserts special code into software so that buffer overflow occurrences can be detected and properly processed such as terminating software execution. The key advantage of such schemes is that they rarely have false positives because they have software execution information. The key limitation of such schemes is that they usually incur an excessive amount of performance overhead because the inserted code needs to be executed for each buffer operation and function call.

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 such existing schemes is that the reported buffer overflow vulnerabilities contain too many false positives fundamentally due to the lack of software execution information and each false positive wastes a huge amount of human effort on manual source code inspection.

Although, several studies have been conducted on application security tools [7], [11], [16]. To date, there are few studies on the effectiveness and efficiency of state-of-the-art static buffer overflow detection techniques. In this paper, we perform an in-depth quantitative and qualitative study on static buffer overflow detection. More specifically, we obtain both the buggy and fixed versions of 100 buffer overflow bugs from 63 real-world projects according to the buffer overflow reports in CVE. Then, quantitatively, we apply `Checkmarx` [3], `Fortify` [10], and `Splint` [22] to all the buggy versions to investigate their false negatives, and also apply them to all the fixed versions to investigate their false positives. We selected those three tools because of the following reasons. First, according to Gartner Group report [11], `Fortify` and `Checkmarx` are leading commercial products in application security market. Second, `Splint` is one of the first open-source tools that concern safety issues in C and is easy to get started with. Therefore, it is also widely used for detecting buffer overflow bugs. We also qualitatively investigate the root reasons for the false-negatives and false-positives of studied techniques to guide the design and implementation of more advanced buffer overflow detection techniques. Finally, we

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.21

also categorized the manual repair patterns of buffer overflow repairs to guide both the manual and automated repair for buffer overflow.

In summary, this paper makes the following contributions:

- A quantitative study of the state-of-art static techniques for buffer overflow detection on 100 real-world bugs from 63 real-world projects totalling 28 Million LoC.
- A qualitative analysis of the false-positives and false-negatives of state-of-the-art static buffer overflow detection techniques, which can guide the design and implementation of more advanced buffer overflow detection techniques.
- A categorization on the fix patterns of buffer overflow bugs to guide both manual and automated buffer overflow repair techniques.

II. Studied Techniques

In this section, we explain the technical details about the techniques that we are going to study. In our study, we intend to study the static analysis tools' capability for detecting buffer overflow vulnerabilities. Our criteria of selecting tools are working on source code, claimed effective in detecting buffer overflow vulnerabilities, and static analysis techniques.

In industry, static analysis technique for buffer overflow vulnerabilities is widely exploited. Commercial static analysis tools that can detect buffer overflow include, for example, HP Fortify [10], Checkmarx [3], Klocwork [15], and Coverity [4]. Also, there are some open source tools for buffer overflow detection, i.e., Splint [22]. These tools have been widely used to help real-world developers and security engineers.

We choose HP `Fortify` and `Checkmarx` as our studied commercial tools. `Splint` is selected as a studied open source tool since it is specifically designed to detect buffer overflow bugs and open-source. In the following subsections, we briefly introduce the techniques employed in these tools.

A. Fortify

HP `Fortify` [10] offers an extensive application security solution. It combines comprehensive static analysis and secure rule management across a multitude of languages and frameworks that allow customers to deploy and get started with quickly. It could be used to find and fix security vulnerabilities and quality problems. It sorts, filters, prioritizes and categorizes the issues found in various forms for the ease of review and analysis.

`Fortify` maintains Fortify Secure Coding Rulepacks which could be extended by user customization. It includes more that 600 secure programming rules and vulnerabilities. It supports more that 20 programming languages, such as ABAP, ASP.NET, C, C++, C# , Classic ASP, COBOL, ColdFusion, Flex/ActionScript, Java, JavaScript/AJAX, JSP, Objective C, PL/SQL, PHP, Python, T-SQL, VB.NET, VBScript, VB6, and XML/HTML. It supports almost all mainstream platforms, e.g., Windows, Linux, Unix, HP-Unix, AIX, Mac OS, and Sun Solaris.

For detecting a buffer overflow, `Fortify` static analysis engine works as follows. First, source code under study is compiled. It is prerequisite for further analysis. Second, `Fortify` conducts data flow analysis by using global, inter-procedural taint propagation analysis to detect the flow of data between a source (site of user input) and a sink (dangerous function call or memory manipulation operation). For example, the data flow analyzer identifies whether a user-controlled input string of unbounded length is being copied into a statically sized buffer. It reports potential vulnerabilities that involve tainted data (user-controlled input) put to potentially dangerous use. Third, `Fortify` conducts control flow analysis and detects potentially dangerous sequences of operations. By analyzing possible execution paths in a program, the control flow analyzer determines whether a set of operations are executed in a certain order. For example, buffer creation, buffer manipulation, and possible buffer overflow. Forth, `Fortify` conducts semantic analysis to detect potentially dangerous uses of functions and APIs at the intra-procedural level. In summary, `Fortify` reports buffer overflow vulnerabilities that involve writing or reading more data than a buffer can hold by conducting data flow analysis, control flow analysis, and semantic analysis.

B. Checkmarx

`Checkmarx` [3] is also a static analysis tool working on source code. `Checkmarx` has strong static analysis features around source code scanning, supports various languages and frameworks, and enables static analysis rule customization. It identifies hundreds of security vulnerabilities in the most prevalent programming languages. `Checkmarx` static code analysis engine offers comprehensive insight into vulnerable patterns and coding flaws. Thus, it can report violations by exposing the applications code properties and code flaws.

For detecting a buffer overflow, `Checkmarx` static code analysis engine scans un-compiled code, un-built code, incomplete code, or even code fragments as well as the entire code base. It creates a meticulous model of how the application interacts with users and other data. It assesses requests and tracks the data and logic flows within the application. It identifies buffer overflow vulnerabilities quickly regarding to the static analysis rules.

C. Splint

`Splint` [8], [17] extends the LCLint to identify the likely buffer overflow vulnerabilities via a static analysis of C program source code. `Splint` employs annotations to specify the programmer assumptions and intents about functions, variables, parameters and types in the source code and standard libraries. The annotations are stylized comments, which are treated as regular C comments by the compiler while recognized as semantic comments by `Splint`. `Splint` can exploit the semantic comments so as to enable local checking of inter-procedural properties during the static analysis. The approach is neither sound nor complete. However, it focuses on lightweight static checking strategies which can achieve good performance and scalability. It can also handle loops with a heuristic manner.

For buffer overflow vulnerability, `Splint` models buffer and annotates buffer sizes in the standard library, such as *strcpy*. During the static analysis, it checks the buffer access to generate precondition and postcondition constraints regarding to buffer bounds. It uses postconditions from previous statements to resolve preconditions. A constraint solver is used to solve the bounds constraints. When the constraints are unsolvable, which means it cannot resolve the preconditions at the beginning of a function, or satisfy the postconditions at the end, a warning is reported.

TABLE I. SUBJECT SYSTEMS

Subjects	Description	Size (LoC)	Language	# BO Bugs(# Versions)
maraDNS	DNS server	81K	C	1(1)
libhx	C library providing data structures	8K	C	1(1)
curl	tool and library for transferring date	170K	C	1(1)
sgminer	GPU miner	51K	C	1(1)
libpng	PNG reference library	54K	C	1(1)
ettercap	comprehensive suite for man in the middle attacks	77K	C	2(2)
ffmpeg	program handling multimedia data	560K	C	6(3)
quagga	network routing software suite	240K	C	1(1)
ntp	network time protocol	266K	C	2(1)
libupnp	UPnP Development Kit	62K	C	3(1)
gnustep-base	software package implementing the API of OpenStep Foundation Kit	1020K	C	1(1)
libwpd	library to help process WordPerfect documents	34K	C++	1(1)
libcgroup	library abstracting the control group file system	18K	C	1(1)
psi	instant messaging application	245K	C++	1(1)
cgminer	miner for bitcoin	91K	C	1(1)
clamav	antivirus engine	878K	C	1(1)
php	general-purpose scripting language	1100K	C	1(1)
Amaya Web Browser	web editor	580K	C	2(1)
inspircd	IRC server	74K	C++	1(1)
bc	numeric processing language	14K	C	1(1)
tiff	support for Tag Image File Format	111K	C	1(1)
sendmail	internetwork email routing facility	134K	C	5(4)
gzip	data compression program	9K	C	1(1)
wireshark	packet analyzer	2748K	C	3(3)
xmp	portable command-line module player	62K	C	2(1)
exim	message transfer agent	142K	C	1(1)
git	version control system	146K	C	1(1)
libxfont	framework providing the core of X11 font system	26K	C	2(2)
csound	sound and music computing system	296K	C	1(1)
freeradius	RADIUS server	147K	C	1(1)
libmms	library for downloading media files	5K	C	1(1)
poppler	PDF rendering library	236K	C++	2(2)
latd	LAT terminal daemon	12K	C++	1(1)
pidgin	chat client	465K	C	1(1)
freetype	library to render fonts	199K	C	6(2)
xvid	video codec library	51K	C	1(1)
binutils	collection of binary tools	1021K	C	2(1)
libproxy	automatic proxy configuration management	5K	C++	1(1)
gimp	GNU Image Manipulation Program	1091K	C	2(2)
icu	library providing unicode and globalization support	754K	C	1(1)
ghostscript	interpreter for the PostScript language and for PDF	1063K	C	3(2)
opensc	tools and libraries for smart cards	116K	C	1(1)
libflac	reference encoder and decoder for FLAC	92K	C	1(1)
openssl	toolkit implementing TLS and SSL protocols	536K	C	1(1)
perl	programming language	773K	C	1(1)
spamdyke	filter for monitoring and intercepting SMTP connections	34K	C	1(1)
dhcp	protocol providing addresses to IP devices	123K	C	1(1)
mapserver	platform for publishing data and applications to the web	276K	C	4(2)
libzip	library for handling zip archives	10K	C	1(1)
man	command used to display user manual	10K	C	3(2)
libthai	Thai language support routines	6K	C	1(1)
graphicsMagick	image processing	414K	C	1(1)
nbd	Network Block Device	3K	C	1(1)
vlc media player	multimedia player	616K	C	1(1)
wu-ftpd	ftp daemon	20K	C	1(1)
squid	caching and forwarding web proxy	287K	C++	2(2)
udisks	D-Bus interfaces to manipulate storage devices	92K	C	1(1)
miniupnpd	software supporting UPnP IGD specifications	9K	C	1(1)
libmodplug	cross-platform MOD decoding library	30K	C++	1(1)
openconnect	SSL VPN client	7K	C	1(1)
bind	program implementing DNS protocols	142K	C	2(2)
sblim-sfcb	project enhancing the manageability of GNU/Linux systems	128K	C	1(1)
openjpeg	JPEG 2000 codec	175K	C	3(2)
Total		28M		100(81)

III. EMPIRICAL STUDY

A. Research Questions

This study aims to investigate the following research questions:

- **RQ1:** How do state-of-the-art static buffer overflow detection techniques perform in terms of false-negatives and false-positives?
- **RQ2:** How do state-of-the-art static buffer overflow detection techniques perform in terms of efficiency?
- **RQ3:** Which types of API or code constructs are closely related to real-world buffer overflow bugs?
- **RQ4:** How do developers manually fix real-world buffer overflow bugs?

In *RQ1* and *RQ2*, we investigate the effectiveness and efficiency of various buffer overflow detection techniques. In *RQ3*, we study the distribution of APIs or code constructs related to buffer overflow. In *RQ4*, we study how developers fix buffer overflow bugs to guide automated buffer overflow repair in the future.

B. *Subject Systems*

To enable objective selection of the buffer overflow bugs, we randomly selected bugs within the buffer overflow category from the CVE website. For each selected bug, we discard it if the corresponding project is not open-source. We continue this process until we find 100 qualified bugs. For each bug, we obtain both the buggy version and repaired version for analysis and inspection. Table I shows the detailed information for the selected subject systems and buffer overflow bugs. In the table, Column "Subjects" lists all the projects that we selected for inspecting buffer overflow bugs; Column "# BO Bugs(# Versions)" list the number of buffer overflow bugs selected from each project and the number of buggy versions. In total, we inspected 100 buffer overflow bugs from the version history of 63 real-world projects, totalling 28 MLoC, ranging from CVE-1999 to CVE-2014.

C. *Experimental Design*

We show our experimental design as follows.

1) Independent Variables: We used the following independent variables (IVs):

IV1: Different Buffer Overflow Detection Techniques. We consider the following 3 widely used buffer overflow detection tools: (1) Checkmarx, (2) Fortify, and (3) Splint.

IV2: Different Buffer Overflow Bugs from Various Projects. We consider 100 different real-world buffer overflow bugs to evaluate the performance of different detection techniques.

2) Dependent Variable: We considered the following dependent variables (DVs):

DV1: False Negatives. We apply all the studied techniques to the buggy versions with buffer overflow bugs to check the set of bugs that are missed by each technique.

DV2: False Positives. We apply all the studied techniques to the fixed versions of buffer overflow bugs to check the set of fixed locations that are mistakenly identified as bugs by each technique.

DV3: Time. We also trace the analysis time for all the studied techniques to evaluate their efficiency.

D. *Experimental Setup*

For each buffer overflow bug, the following steps are performed:

First, we apply all the studied techniques to analyze the corresponding faulty version to find the bugs that cannot be detected for each technique, i.e., false-negatives. In addition, we record the analysis time for each tool.

Second, we apply all the studied techniques to analyze the corresponding fixed version to find the fixed bugs that are still identified as bugs, i.e., false-positives. In addition, we record the analysis time for each tool.

Finally, we perform qualitative analysis on each bug: (1) we record the detailed bug information (e.g., the API involved); (2) we manually analyze the reason for the false-positives and false-negatives of the studied tools; (3) we categorize the manual fix pattern for the bug.

We repeat the steps for all the 100 studied bugs. All our Fortify and Splint experiments were performed on a server with Intel Xeon CPU E5-2603 (1.80GHz) and 128GB RAM on Ubuntu Linux 12.04 and Checkmarx on a server with Intel Xeon CPU E5-2650 (2.30GHz) and 384GB RAM on Windows Server 2008. We use the second server because our Checkmarx's license is tied to it.

E. *Result Analysis*

In this section, we present the detailed results for our empirical study.

1) RQ1: Effectiveness for Buffer Overflow Detection: We applied Checkmarx, Fortify and Splint to all buggy and fixed versions to investigate their false negatives and false positives. Note that not all the 100 bugs can be successfully analyzed by the three studied techniques. For example, Fortify needs to compile the source code in order to do the analysis. Also, we set a 3-hour time limit because when the source code is too large, it may take hours to do the analysis. We treat this as not being able to reveal any bug. Both Fortify and Checkmarx face this problem. For any technique, if the analysis time reaches the 3-hour timeout limit, we treat the tool as successfully applied to the case, but failing to report any bug. Checkmarx doesn't require the code compilation. Instead, it scans the source code and directly applies static analysis rules on the code files (it works for incomplete code). Therefore, it can be applied to all cases. When using Fortify, it can't be applied if the program can't be compiled. When using Splint, we met various preprocessing and parsing errors, and also cannot successfully apply it for all cases. Finally, Fortify, Checkmarx, and Splint were successfully applied to 60, 100 and 23 cases, respectively.

Table II shows the overall repair results. In the table, Column "Techs" list all the applied techniques; Column "# Identified Bugs" lists the number of bugs found by each technique for the buggy version before repair; Column "FN Rate" presents the false negative rates for the studied techniques, i.e., the ratio of the number of bugs which cannot be found by the studied techniques to the number of cases to which the technique can be applied successfully; Column "# Identified Fixes" presents the number of buffer overflow bugs identified as fixed by the studied techniques on the repaired version; Column "FP Rate" presents the false positive rates for the studied techniques, i.e., the ratio of the number of the fixed bugs that are still identified as bugs to the number of all the corresponding buggy versions detected by the studied techniques. Note that the first three rows represent Fortify, Checkmarx, and Splint, respectively. In addition, in practice, the user can use any combination of the existing techniques together to detect potential buffer overflow bugs. Therefore, we further investigate the effectiveness of applying two or all of the three studied techniques together, which is shown in the last four rows of the table. For example, Fortify+Checkmarx denotes that we treat Fortify and Checkmarx as a whole – the combined technique reports a bug when any of the two techniques reports a bug, while treating the project under test as bug free if none of the two techniques can detect a bug. From the table, we have the following observations:

First, the Checkmarx technique is able to find the most buffer overflow bugs, followed by the Fortify technique. In total, the Checkmarx technique is able to find 32 bugs, while the Fortify and Splint techniques are only able to find 19 and 10 bugs, respectively. We think the reason why Checkmarx performs the best is that (1) Checkmarx does not require compiled code, and thus can work for more case; and (2) its powerful static analysis engine includes comprehensive buffer overflow detection rules. The reason why Splint performs the worst is that it can only be applied to a small ratio of cases which do not have preprocessing errors or

TABLE II. OVERALL REPAIR RESULTS

Techs	# Identified Bugs	FN Rate	# Identified Fixes	FP Rate
Fortify	19	68.3% (41/60)	13	31.6% (6/19)
Checkmarx	32	68.0% (68/100)	8	75.0% (24/32)
Splint	10	56.5% (13/23)	0	100.0% (10/10)
Fortify+Checkmarx	42	58.0% (58/100)	14	66.7% (28/42)
Checkmarx+Splint	39	61.0% (61/100)	7	82.1% (32/39)
Fortify+Splint	26	59.4% (38/64)	13	50.0% (13/26)
All	47	53.0% (53/100)	13	72.3% (34/47)

parse errors. Furthermore, the false negative rates for all the studied techniques are close, e.g., Checkmarx and Fortify share similar false negative rates, while Splint has a slightly lower false negative rate. The reason is that Splint is a lightweight technique that simply report almost all the possible bugs and may be imprecise.

Second, in terms of false positive rates, Fortify tends to perform the best, while Splint performs the worst. For example, for all the 19 bugs identified by Fortify for the buggy versions, it successfully reported 13 as repaired. That is, only 31.6% (i.e., 6 out of the 19) of the bug-free cases are mistakenly reported as buggy (false positives). On the contrary, for all the 10 bugs identified by Splint for the buggy versions, it was not able to identify any fix, i.e., the false positive rate is 100% for Splint. We think the reason is that Splint trades off precision for scalability. According to [16], when Splint deals with complex situation (increased complexity of index, address or length, more complex containers and flow control constructs), the loss of precision leads to increased false alarms (false positive rate). On the other hand, the compilation gives Fortify much information to identify the fix.

Third, the combination of different techniques can bring non-trivial benefits in terms of false negative rates. For example, a single tool can at most identify 32 bugs (i.e., Checkmarx); when additionally using Fortify, we can identify 10 more bugs; when further using Splint, we can identify 47 bugs, which is nearly half of all studied bug! Of course, lower false negative rates may incur higher false positive rates. For example, the false positive rate is 31.6% when using Fortify alone, while it becomes 72.3% when using all the three techniques together. However, when finding more bugs (low false negative rate) has higher priority, combining tools will meet users' requirements.

In summary, we find Checkmarx to be the tool that can detect the most overflow bugs among the three tools. Also, we find that Splint performs the best in terms of false negative rate, while Fortify performs the best in terms of the false positive rate. In practice, we encourage the users to use all the three tools together to achieve ideal performance in terms of false negative rates, and use the Fortify tool alone to achieve ideal performance in terms of false positive rates.

2) RQ2: Efficiency for Buffer Overflow Detection: We further record the analysis time of the studied techniques to compare their efficiency. As we showed in Section III-E1, each technique has some limitation and can only be successfully applied to a subset of the studied bugs. To enable a fair comparison, we used all the 15 bugs where all the three techniques can successfully apply. To further compare the analysis time of Fortify and Checkmarx, we listed the remaining 32 bugs where both Fortify and Checkmarx can apply. The detailed results are shown in Table III. In the table, Column "Subjects" lists the corresponding subject and version information for the 47(15+32) bugs; Columns CVE ID shows the CVE ID of this bug; Columns 3 and 4 present the analysis time by Fortify on the buggy and fixed versions for each bug, respectively. Similarly, Columns 5 and 6 present the corresponding analysis time for Checkmarx, while Columns 7 and 8 present the corresponding analysis time for Splint. From the table, we have the following observations:

First, Fortify needs to compile the source code and it gets much more information than Checkmarx. So it is reasonable to predict that Fortify needs longer time than Checkmarx to analyze the program. However, from the table, we can see that this hypothesis does not always hold. Actually, Checkmarx consumes more time than Fortify for 32 of the 47 bugs when applied to the buggy versions. The total analysis time for buggy versions is 46344 seconds for Checkmarx, while only 31591 seconds for Fortify. Furthermore, in some cases, Checkmarx's analysis time is about 7 times longer than Fortify. The reason behind this finding is also the compilation process. Not all the C files in the source code are compiled. Sometimes the source code contains some C files for other platform or testing. When we configure and compile the source code, these C files are omitted. Therefore, Fortify does not analyze those files. However, Checkmarx doesn't compile the source code, and it scans all files in it. So Checkmarx scans much more files than Fortify in some cases. This is the main reason that why in these cases Checkmarx's analysis time is 7X longer than Fortify.

Second, Splint runs much faster than Fortify and Checkmarx on all these 15 bugs. While it takes minutes to hours to analyze a bug using Fortify and Checkmarx, it only takes seconds using Splint. There are several potential reasons: (1) Splint uses lightweight static checking strategies which is neither sound nor complete, and it trades off precision for scalability; (2) When applied to a whole project, Splint skips a large number of files when it cannot process them due to the preprocessing and parsing errors.

In summary, the Checkmarx technique tends to be the most costly technique to apply, followed by the Fortify technique.

3) RQ3: Buffer Overflow Bug Distribution: Each buffer overflow bug is related to some specific APIs or code constructs. For example, it can be an *array* crossing its boundary, or the *memcpy* API with a target buffer of insufficient size. Understanding those APIs or code constructs is essential for detection and understanding of buffer overflow bugs. Therefore, we manually inspected each of the 100 studied bugs to identify their related APIs or code constructs. The first two columns in Table IV presents the distribution of APIs or code constructs related to buffer overflow bugs. In the table, Column "API" lists all the APIs or code constructs related

TABLE III. ANALYSIS TIME STATISTICS

Subjects	CVE ID	Fortify		Checkmarx		Splint	
		Buggy	Repaired	Buggy	Repaired	Buggy	Repaired
bc-1.06	N/A	134s	89s	91s	91s	2s	2s
gzip-1.2.4	CVE-2001-1228	79s	82s	60s	92s	3s	4s
squid-2.4.STABLE6	CVE-2002-0713	462s	460s	740s	741s	38s	39s
openjpeg-1.5.0	CVE-2012-3535	220s	221s	1417s	1289s	26s	25s
openjpeg-1.5.0	CVE-2012-3358	220s	221s	1417s	1289s	26s	25s
openjpeg-1.4.1	CVE-2012-1499	236s	220s	745s	1417s	16s	17s
openssl-1.0.1	CVE-2012-2110	4060s	3900s	5726s	6000s	371s	380s
libpng-1.5.9	CVE-2011-3048	111s	109s	273s	273s	9s	9s
xmp-2.5.1	CVE-2007-6731	963s	950s	630s	595s	19s	17s
xmp-2.5.1	CVE-2007-6732	963s	950s	630s	595s	19s	17s
maraDNS-1.4.05	CVE-2011-0520	371s	372s	532s	532s	17s	17s
libthai-0.1.12	CVE-2009-4012	52s	50s	91s	91s	1s	1s
libhx-3.5	CVE-2010-2947	80s	80s	91s	91s	3s	3s
udisks-2.1.2	CVE-2014-0004	298s	300s	1412s	1533s	50s	55s
libcgroup-0.37	CVE-2011-1006	132s	151s	157s	212s	3s	3s
sendmail-8.12.7	CVE-2002-1337	795s	787s	615s	584s	–	–
sendmail-8.11.5	CVE-2001-0653	787s	779s	405s	406s	–	–
sendmail-8.12.4	CVE-2002-0906	707s	714s	585s	584s	–	–
sendmail-8.12.7	N/A	795s	787s	615s	584s	–	–
man-1.5h1	CVE-2001-0641	230s	230s	144s	145s	–	–
man-1.5i2	CVE-2001-1028(1)	230s	238s	145s	144s	–	–
man-1.5i2	CVE-2001-1028(2)	230s	238s	145s	144s	–	–
freetype-2.4.8	CVE-2012-1126	708s	748s	1270s	1270s	–	–
freetype-2.4.8	CVE-2012-1132	708s	748s	1270s	1270s	–	–
freetype-2.4.8	CVE-2012-1134	708s	748s	1270s	1270s	–	–
freetype-2.4.8	CVE-2012-1135	708s	748s	1270s	1270s	–	–
freetype-2.4.8	CVE-2012-1140	708s	748s	1270s	1270s	–	–
inspircd-2.0.5	CVE-2012-1836	501s	508s	610s	612s	–	–
freetype-2.3.12	CVE-2010-3311	954s	716s	1241s	1241s	–	–
sblim-sfcb-1.3.7	CVE-2010-1937	945s	964s	1161s	1191s	–	–
git-1.7.1	CVE-2010-2542	956s	975s	1177s	1273s	–	–
nbd-2.9.19	CVE-2011-0530	67s	63s	60s	60s	–	–
libupnp-1.6.17	CVE-2012-5958	298s	295s	310s	310s	–	–
libupnp-1.6.17	CVE-2012-5959	298s	295s	310s	310s	–	–
libupnp-1.6.17	CVE-2012-5960	298s	295s	310s	310s	–	–
latd-1.30	CVE-2013-0251	111s	115s	123s	123s	–	–
cgminer-4.3.4	CVE-2014-4501	790s	790s	1127s	1188s	–	–
sgminer-4.2.1	CVE-2014-4501	502s	522s	860s	765s	–	–
libxfont-1.4.6	CVE-2013-6462	312s	313s	249s	218s	–	–
freeradius-3.0.1	CVE-2014-2015	804s	1171s	1167s	1362s	–	–
libmms-0.6.3	CVE-2014-2892	74s	75s	60s	60s	–	–
libflac-1.3.0	CVE-2014-8962	1281s	1303s	1533s	1098s	–	–
tiff-4.0.3	CVE-2013-1961	612s	609s	920s	805s	–	–
squid-3.3.6	CVE-2013-4115	1777s	1735s	3289s	2996s	–	–
poppler-0.24.1	CVE-2013-4473	1726s	1711s	4755s	5678s	–	–
poppler-0.22.0	CVE-2013-1788	1718s	1695s	3630s	3501s	–	–
libmodplug-0.8.8.1	CVE-2011-1574	2468s	2470s	436s	369s	–	–
Average		672.15s	681.23s	986.04s	976.00s	40.20s	40.93s

TABLE IV. API RELATED DETECTING RESULTS

API	# Instances	Fortify	Checkmarx	Splint
array	31	4/21	2/31	8/12
memcpy	15	0/5	9/15	0/1
sprintf	13	6/7	7/13	1/2
pointer	7	0/4	0/7	0/1
strcpy	6	4/4	3/6	0/1
strncpy	5	1/4	1/5	0/0
malloc	4	1/2	2/4	0/1
sscanf	4	1/3	3/4	0/2
memset	3	1/2	1/3	1/1
strcat	2	0/1	2/2	0/0
memmove	1	0/1	0/1	0/0
strncmp	1	0/1	0/1	0/0
others	8	1/5	2/8	0/2
Total	100	19/60	32/100	10/23

to buffer overflow bugs; Column ”# Instance” presents the number of bugs related to a specified API. From these two columns, we observe that *array*, *memcpy*, and *sprintf* account for 31%, 15%, and 13% of buffer overflow bugs, and are the top three APIs or code constructs related to buffer overflow bugs. On the contrary, *strcat*, *memmove*, and *strncmp* are the three APIs with least number of buffer overflow bugs. *Array* is almost used in every C program, so it comes the first. Memory manipulation and string operation are also very common in C program. *Memcpy* is widely used in memory manipulation and *sprintf* is the first choice when it comes to string operation. So *memcpy* and *sprintf* comes the second and third.

This distribution provides valuable practical guidance in various aspects: (1) it reminds programmers to be careful when using those error-prone APIs or code constructs; (2) the static analysis techniques can take advantage of such distribution and prioritize the checking points by mainly focusing on the error-prone APIs or code constructs when the analysis time is limited or the tools are too costly.

We also found out the relation between APIs and identified bugs for each tool, which is shown in the last three columns of Table IV. These three columns show the number of identified bugs for each tool on some specified API and the number of bugs the tool can apply to. For example, 4/21 in the table means when the API is *array*, Fortify can be applied to 21 bugs and identify 4 of them.

We notice that although Splint can only be applied to 12 bugs when the API is *array*, it identifies most buffer overflow bugs, 8 out of 12. For Fortify and Checkmarx, it is only 4/21 and 2/31, respectively. We think the reason behind this

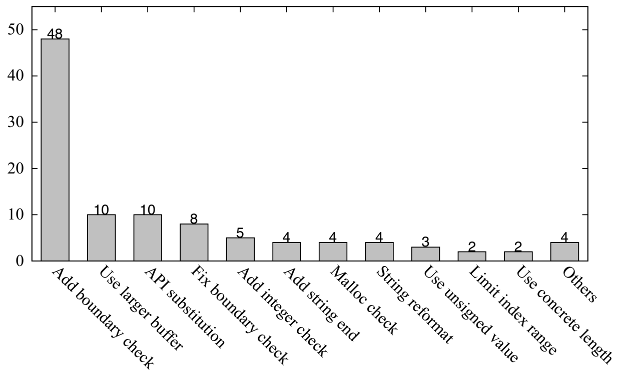

Fig. 1. Fix strategies for buffer overflow bugs

is that in real-world projects, the index of an *array* tends to be complex and *array* is almost used in every function. For commercial tools like `Fortify` and `Checkmarx`, they need stronger evidence to report an *array* as a buffer overflow, because otherwise there will be too many false alarms. On the other hand, `Splint` uses a constrain solver to solve the bounds constrain. If the constrain is unsolvable, which is a common scene in real-world programs, it reports a warning.

For `Fortify` and `Checkmarx`, although they do not work well on *array*, it is a different story when it comes to APIs like *sprintf* and *strcpy*. For `Fortify`, it is 6/7 on *sprintf* and 4/4 on *strcpy*. `Fortify` almost finds all of them. For `Checkmarx` it is 7/13 and 3/6, which is also acceptable. We think the reason is that *sprintf* and *strcpy* are both unsafe API. The safer versions are *snprintf* and *strncpy*, which use a parameter *n* to control how many bits should be copied. When `Fortify` and `Checkmarx` find these unsafe APIs, they tend to report a warning, because these APIs are not widely used as *array* across the project and they are error-prone. The row for *strncpy* in the table, which is the safer version of *strcpy*, is 1/4 for `Fortify` and 1/5 for `Checkmarx`. This supports our finding because they find fewer bugs on safe API.

In summary, *array*, *memcpy* and *sprintf* are the top three APIs related to buffer overflow bugs. `Splint` works well on array, where `Fortify` and `Checkmarx` need to improve their performance. However, `Fortify` and `Checkmarx` can find most bugs on unsafe APIs like sprintf and strcpy.

4) RQ4: Repair Strategy Categorization: In Section III-E3, we studied the distribution of studied buffer overflow bugs, which may help with the automated detection and understanding of buffer overflow bugs. In this section, we further study how the buffer overflow bugs are fixed by the developers. The manual fix strategies can help with the automated or manual fixing of buffer overflow bugs. The overall results are show in Fig 1. In the figure, the horizonal axis presents the manual strategies for fixing the buffer overflow bugs, the vertical axis present the number of fixed bugs, and the gray bars presents the number of buffer overflow bugs fixed by each strategy. From the figure, we can find that almost a half of all the buffer overflow bugs can be fixed by adding boundary checks. In addition, using larger buffers and API substitution both account for 10% of all bugs. We next present a detailed explanation for all those common strategies used by the developers.

Add boundary check. Adding a boundary check before a *strcpy* or other API can avoid the buffer overflow (e.g., Listing 1). This is the most widely used strategy for fixing the studied buffer overflow bugs, indicating that missing boundary checks can be the main reason for buffer overflow bugs.

Listing 1. Add boundary check

```
/*buggy version: gzip-1.2.4
  file: gzip.c*/
  1009 strcpy(ifname, iname);

/*fixed version: gzip-1.3.9
  file: gzip.c*/
  1055 if (sizeof ifname - 1 <= strlen(iname))
  1056   goto name_too_long;
  1057
  1058 strcpy(ifname, iname);
```

Use larger buffer. Copying too many bytes to a buffer leads to a buffer overflow, and using a larger buffer can fix this to some degree. For example, as shown in Listing 2, increasing the size of the overflowed buffer can potentially fix the corresponding bug.

Listing 2. Use larger buffer

```
/*buggy version: ffmpeg-0.7.11
  file: libavcodec/kgv1dec.c*/
  42 int offsets[7];
   ...
  87 int odix = (code >> 10) & 7;
   ...
  92 if (offsets[odix] < 0){
   ...
  97 }

/*fixed version: ffmpeg-0.7.12
  file: libavcodec/kgv1dec.c*/
    49 int offsets[8];
     ...
    98 int odix = (code >> 10) & 7;
     ...
  103 if (offsets[odix] < 0){
     ...
  108 }
```

API substitution. Substituting another safer API for the buggy one is also widely used to fix buffer overflow bugs. For example, as shown in Listing 3, *sprintf* can be substituted by *snprintf* because *snprintf* has one more parameter 'MaxMsgLength' which will limit the bytes written into msgBuffer.

Listing 3. API substitution

```
/*buggy version: Amaya Web Browser-11.0.1
  file: amaya/Xml2thot.c*/
  3297 if (val <= 0)
  3298 {
  3299   sprintf ((char *)msgBuffer, "Unknown attribute value
       \"%s\"", (char *) attrValue);
      ...
  3302 }

/*fixed version: Amaya Web Browser-11.2
  file: amaya/Xml2thot.c*/
  3312 if (val <= 0)
  3313 {
  3314   snprintf ((char *)msgBuffer, MaxMsgLength, "Unknown
       attribute value \"%s\"", (char *) attrValue);
       ...
  3317 }
```

Fix boundary check. Sometimes the programmer was aware of the potential buffer overflow and added boundary check to prevent this. However, the boundary check can be incorrect or imprecise, e.g., off-by-one error. In this case, fixing the boundary check can fix the corresponding bug as shown in Listing 4.

Listing 4. Fix boundary check

```
/*buggy version: freetype-2.4.8
 file: src/truetype/ttinterp.c*/
 7522 if(...)
 7523 {
 7524    if (CUR.IP + 1 > CUR.codeSize)
 7525      goto LErrorCodeOverflow;
 7526
 7527    CUR.length = 2 - CUR.length * CUR.code[CUR.IP + 1];
 7528 }

/*fixed version: freetype-2.4.9
 file: src/truetype/ttinterp.c*/
 7545 if(...)
 7546 {
 7547   if (CUR.IP + 1 >= CUR.codeSize)
 7548      goto LErrorCodeOverflow;
 7549
 7550    CUR.length = 2 - CUR.length * CUR.code[CUR.IP + 1];
 7551 }
```

Add integer check. Integer overflow is a common cause of buffer overflow, checking the involved integer can prevent this issue. For example, as shown in Listing 5, the bug fix adds an integer check to force the execution to terminate if the integer can overflow.

Listing 5. Add integer check

```
/*buggy version: openssl-1.0.1
 file: crypto/asn1/a_d2i_fp.c*/
 172 if(!BUF_MEM_grow_clean(b,len+want)
 173 {
 174   ASN1err(ASN1_F_ASN1_D2I_READ_BIO,ERR_R_MALLOC_FAILURE
     );
 175   goto err;
 176 }

/*fixed version: openssl-1.0.1a
 file: crypto/asn1/a_d2i_fp.c*/
 167 if(len + want < len || !BUF_MEM_grow_clean(b,len+want))
 168 {
 169   ASN1err(ASN1_F_ASN1_D2I_READ_BIO,ERR_R_MALLOC_FAILURE
     );
 170   goto err;
 171 }
```

Add string end. All string should end with string end '\0'. Otherwise copying it may lead to a buffer overflow. Adding string end to the end of the string can solve the problem (shown in Listing 6).

Listing 6. Add string end

```
/*buggy version: latd-1.30
 file: llogincircuit.cc*/
 94 char error[1024];
  ...
 96 sprintf(error, "llogin version %s does not match latd
       version " VERSION, cmdbuf);

/*fixed version: latd-1.31
 file: llogincircuit.cc*/
   94 char error[1024];
    ...
   97 if (len > 900)
   98   len = 900;
   99 cmdbuf[len] = '\0';
    ...
 101 sprintf(error, "llogin version %s does not match latd
       version " VERSION, cmdbuf);
```

Malloc check. The use of failed *malloc* may cause the buffer overflow problem. Therefore, checking the corresponding *malloc* parameter and return value can be used to avoid buffer overflow bugs, as shown in Listing 7.

Listing 7. Malloc check

```
/*buggy version: ffmpeg-0.7.11
 file: libavformat/nsvdec.c*/
 311 p = strings = av_mallocz(strings_size + 1);
 312 endp = strings + strings_size;
 313 avio_read(pb, strings, strings_size);
 314 while (p < endp) {
     ...
     }

/*fixed version: ffmpeg-0.7.12
 file: libavformat/nsvdec.c*/
 311 p = strings = av_mallocz(strings_size + 1);
 312 if (!p)
 313   return AVERROR(ENOMEM);
 314 endp = strings + strings_size;
 315 avio_read(pb, strings, strings_size);
 316 while (p < endp) {
     ...
     }
```

String reformat. Formatting string can control how many bytes are written to a buffer, e.g. changing from *sscanf(a,"%s", buf)* to *sscanf(a,"%10s", buf)* can ensure that 10 bytes are written to *buf* at most. Listing 8 presents an example bug fix. In this example, the formatting string of *sscanf* is changed from "%d%d%s" to "%d%d%ns". Here, n is the value of "sizeof(endbuf)-1".

Listing 8. String reformat

```
/*buggy version: gimp-2.6.11
 file: plug-ins/common/sphere-designer.c*/
 1992 gchar endbuf[21*(G_ASCII_DTOSTR_BUF_SIZE + 1)];
 1993 gchar *end = endbuf;
       ...
 2029 if (sscanf (line, "%d %d %s", &t->majtype, &t->type,
       end) != 3)
      {
        ...
      }

/*fixed version: gimp-2.6.12
 file: plug-ins/common/sphere-designer.c*/
 1992 gchar endbuf[21*(G_ASCII_DTOSTR_BUF_SIZE + 1)];
 1993 gchar *end = endbuf;
       ...
 1995 gchar fmt_str[16];
       ...
 2020 snprintf (fmt_str, sizeof (fmt_str), "%%d %%d %%%lds",
       sizeof (endbuf)-1);
       ...
 2032 if (sscanf (line, fmt_str, &t->majtype, &t->type,
       end) != 3)
      {
        ...
      }
```

Listing 9. Use unsigned value

```
/*buggy version: xmp-2.5.1
 file: src/misc/oxm.c*/
 52 int ilen;
   ...
 79 for (i = 0; i < nins; i++){
 80   ilen = read32l(f);
 81   if (ilen > 263)
 82     return -1;
     ...
   }

/*fixed version: xmp-2.6.2
 file: src/misc/oxm.c*/
 51 uint32 ilen;
   ...
 78 for (i = 0; i < nins; i++){
 79   ilen = read32l(f);
 80   if (ilen > 263)
 81     return -1;
     ...
   }
```

Use unsigned value. As shown in Listing 9, if a boundary check is something like *if(*$len > 256$*)*, and variable len is an integer, then assigning -1 to len will bypass the boundary check and cause a buffer overflow when len is used as a parameter of a function call like *memcpy*. The reason is that *memcpy* needs an unsigned parameter and will take value -1 as a large positive integer. This leads to a potential buffer overflow.

Limit index range. Limit the buffer index range can also fix buffer overflow bugs. For example, as shown in Listing 10, the *MIN* function is defined as $MIN(a, b) = a < b?a : b$, and $index = min(index, MAXIMUM)$ can be used limit the value of index to be no greater than MAXIMUM so as to avoid buffer overflow bugs.

Listing 10. Limit index range

```
/*buggy version: opensc-0.11.13
 file: src/libopensc/card-acos5.c*/
 143 memcpy(card->serialnr.value, apdu.resp, apdu.resplen);

/*fixed version: opensc-0.12.0
 file: src/libopensc/card-acos5.c*/
 141 memcpy(card->serialnr.value, apdu.resp,
      MIN(apdu.resplen, SC_MAX_SERIALNR));
```

Use concrete length. In a function call like *memcpy(buf,data,len)*, the parameter len could lead to a buffer overflow. If we know that only first 4 octets of *buf* is used, and substituting '4' for '$data.len$' can solve the problem. See Listing 11.

Listing 11. Use concrete length

```
/*buggy version: dhcp-4.1.0
 file: client/dhclient.c*/
 3057 memcpy (netmask.iabuf, data.data, data.len);

/*fixed version: dhcp-4.2.0
 file: client/dhclient.c*/
 3088 memcpy (netmask.iabuf, data.data, 4);
```

We further investigated the repair strategies for each type of buffer overflow bugs. The detailed results are shown in Table V. In the table, Column "Strategies" present all the manual repair strategies for the studied bugs; the other columns list all the types of buffer overflow bugs according to their related code APIs or constructs. Then, each cell presents the number of corresponding type of bugs fixed by the corresponding repair strategy. Note that sometimes the developers use more than one strategies to fix one bug. According to the table, we have the following observations.

First, the "Add boundary check" strategy is able to fix the majority types of buffer overflow bugs. The only exceptions are for *malloc*, *sscanf*, and *memmove*. We only explain the reason for *malloc* and *sscanf* because there is only one instance of *memmove* and may not be representative. For *malloc*, the "Malloc check" strategy is used. A buffer overflow may occur if the parameter and return value of malloc is not checked. "Malloc check" checks them and ensures that no buffer overflow will happen. For *sscanf*, "String reformat" strategy is the easiest way to fix it.

Second, although "Add boundary check" can fix the most bugs overall, it is not always the case for specific types of bugs. For example, *sprintf* bugs are more likely to be fixed by the "API substitution" strategy rather than the "Add boundary check" strategy; *sscanf* bugs are more likely to be fixed by "String reformat"; *memset* bugs are more likely to be fixed by "Add integer check". Substituting *snprintf* for *sprintf* is a convention, because *snprintf* is a safer version of *sprintf*. For *sscanf*, the reason is explained before, which is "String reformat" being the easiest way to fix it.

> In summary, the results show that adding boundary check can fix nearly half of the studied bugs. However, for each specific type of bugs, there may be more suitable way to fix those bugs. This provide practical guidelines for both manual and automated repair of buffer overflow bugs. For example, an effective manual/automated repair strategy would first try if adding boundary checks can resolve the bug. If the first step fails, the strategy can then try different actions according to the APIs involved in the buffer overflow, e.g., a buffer overflow bug involving the *malloc* API can usually be fixed using *malloc check*.

F. *Threats to Validity*

The threat to internal validity lies in the intensive manual inspection and data analysis performed in the empirical study. To reduce this threat, the first author together with other two graduate students reviewed all the manual inspection results to guarantee the precision. In addition, we also reviewed all the data analysis scripts to ensure the correctness.

The threats to external validity mainly lie in the subjects and bugs used in the study. All the 63 subjects used in the empirical study are real-world projects, and come from various application domains. However, due to the intensive manual inspection, we were only able to study 100 buffer overflow bugs, which may not be sufficiently representative. Moreover, for investigating the false positives of the studied techniques, we only considered the fixed locations of the 100 bugs. The reason is that there is no ground truth for other instances reported by the studied techniques. This threat can be further reduced by using more buffer overflow bugs from more real-world projects in the future.

The threats to construction validity lie in the metric used to assess the effectiveness and efficiency of the studied techniques. To reduce this threat, we used the widely used metrics, such as false positive rate and false negative rate, to evaluate the studied techniques.

IV. Related Work

Research on buffer overflow detection mainly focus on proposing new techniques for buffer overflow detection. In addition, researchers also performed various studies on buffer overflow detection. In this section, we discuss about related work in both directions.

A. *Buffer Overflow Detection Techniques*

The dynamic approaches [5], [12], [14], [19], [30] require program execution to identify potential buffer overflow bugs. A vast majority of dynamic buffer overflow detection techniques insert special code into software so that buffer overflow occurrences can be detected and properly processed such as terminating software execution. For example, some dynamic analysis tools [8], [25] check whether the return addresses of function invocations have been modified to detect buffer overflow attacks. Some other techniques [14], [19], [28] assume the boundary of variables should not be exceeded by all accesses, and use this heuristic to identify potential buffer overflow bugs. Although those dynamic techniques do not suffer from the false positive problem, it is hard to generate test inputs to expose the buffer overflow bugs. Actually, the

TABLE V. DETAILED FIX STRAGEGIES

Strategies	array	memcpy	sprintf	pointer	strcpy	strncpy	malloc	sscanf	memset	strcat	memmove	strncmp	others
Add boundary check	18	9	2	4	5	5	–	–	1	1	–	1	4
Use larger buffer	3	1	4	1	–	–	–	–	–	–	–	–	1
API substitution	–	1	6	–	2	–	–	–	–	1	–	–	–
Fix boundary check	5	–	–	1	–	–	–	–	–	–	–	–	–
Add integer check	2	–	–	–	–	–	–	–	2	–	1	–	–
Add string end	–	–	3	–	–	–	–	1	–	–	–	–	–
Malloc check	–	–	–	–	–	–	4	–	–	–	–	–	–
String reformat	–	–	–	–	–	–	–	3	–	–	–	–	1
Use unsigned value	1	1	–	–	–	–	–	–	–	–	–	–	1
Limit index range	1	1	–	–	–	–	–	–	–	–	–	–	–
Use concrete length	–	1	–	–	–	–	–	–	–	–	–	–	1
Others	2	2	–	1	–	–	–	–	–	–	–	–	1

users mainly use random or manual testing to identify buffer-overflow-triggering test inputs. To systematically generate test inputs to expose buffer overflow bugs, Splat [30] is proposed to automatically generate test cases for detecting buffer overflow bugs. Splat performs directed random testing guided by symbolic execution. As symbolic execution can be extremely expensive, Splat also proposes the symbolic length abstraction technique to prune the search space of symbolic execution without losing the buffer overflow detection ability. In spite of the optimizations embodied by Splat, it is still an expensive technique and not widely used in practice. It is hard to compare dynamic techniques for buffer overflow detection since they either require manual/random test generation, or can be extremely expensive to apply.

The static program analysis approaches [1], [2], [8], [18], [20], [21], [23]–[27], [29], [31], [32] scan software source code under test to discover potential code segments that are vulnerable to buffer overflow attacks. As the static approaches does not execute the program under analysis, there can be many false alarms. Therefore, each reported vulnerability warning requires further manual checking. A number of tools (e.g., ITS4 [26] and FlawFinder [31]) scans C or C++ source code, breaks the codes into lexical tokens, and then matches patterns in the token stream to find possible buffer overflow bugs. Although simple and scalable to large scale programs, these techniques only consider the lexical information and may not be effective. Some other tools further perform sematic analysis. `Fortify` [10] reports buffer overflow vulnerabilities that involve writing or reading more data than the buffer capacity by conducting data flow analysis, control flow analysis, and semantic analysis all together. `Checkmarx` [3] creates a meticulous model of how the application interacts with users and other data. Then, `Checkmarx` assesses requests and tracks the data and logic flows within the application to identify buffer overflow vulnerabilities based on various static analysis rule. `Splint` [17] uses several lightweight static analysis techniques. It requires users to write source annotations to apply inter-procedural analysis. For buffer overflow, `Splint` models and annotates buffer sizes in standard libraries, such as *strcpy*, to report potential warnings for all library functions susceptible to buffer overflow vulnerabilities. In this work, we focus on studying the effectiveness and efficiency of state-of-the-art static techniques for buffer overflow detection.We choose `Fortify` and `Checkmarx` as the studied commercial tools and `Splint` as the studied open source tool.

B. Studies on Buffer Overflow Detection

Wilander and Kamkar [28] performed an empirical study on dynamic buffer overflow detection techniques using 20 different buffer overflow bugs. The study shows that even the best dynamic tool can only detect 50% bugs, and there are 6 bugs cannot be detected by any studied tool. In contrast, our study is on static analysis techniques for buffer overflow detection, and is conducted using much more real-world buffer overflow bugs. Recently, Fang and Hafiz [9] performed an empirical study on reporters of buffer overflow vulnerabilities to understand the detection tools and methods. They found that most reporters mainly use fuzzing, and static analysis tools are rarely used. Our study shows that nearly half of real-world buffer overflow bugs can be detected by state-of-the-art static analysis tools, demonstrating the effectiveness of static analysis tools in practice. Johnson et al. [13] studied the static analysis techniques in general, and found that false positives are the barriers to use static analysis techniques in practice. In this paper, we studied the static analysis techniques for buffer overflow detection, and the results of our controlled experiments show that static analysis techniques (e.g., `Fortify`) can achieve acceptable false positive rate for detecting buffer overflow bugs.

V. CONCLUSION

Although various buffer overflow detection techniques have been proposed, there are few studies on comparing the effectiveness and efficiency of state-of-the-art static analysis techniques. In this paper, we present an empirical study on both the detection and fixing of buffer overflow bugs. More specifically, we compared the efficiency and effectiveness of the `Fortify`, `Checkmarx`, and `Splint` tools on 100 buffer overflow bugs from 63 real-world projects totalling 28 MLoC. In addition, we also investigated the distribution of buffer overflow bugs, as well as the fixing strategies for different buffer overflow bugs. The results provide practical guidelines on detecting, understanding, as well as fixing the buffer overflow bugs. In the future, we will extend our study in the following directions: more static analysis techniques or dynamic testing techniques, other critical vulnerabilities in the area of application security, more real world projects.

ACKNOWLEDGMENTS

The paper was partially supported by the National Grand Fundamental Research 973 Program of China (No.2014CB340703) and the National Natural Science Foundation of China (No. 91318301, 61472179, 61561146394). We would like to express our gratitude to CHECKMARX China team for providing an evaluation version of `Checkmarx` to support our comparison study. We thank all the students who helped us in the experiment.

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “Aeg: Automatic exploit generation.” in *Network and Distributed System Security Symposium (NDSS)*, vol. 11, 2011, pp. 59–66.

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing symbolic execution with veritesting,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014, pp. 1083–1094.

[3] “Checkmarx homepage,” https://www.checkmarx.com (accessed January, 2016).

[4] “Coverity homepage,” http://www.coverity.com/ (accessed January, 2016).

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” in *USENIX Security*, vol. 98, 1998, pp. 63–78.

[6] “CVE homepage,” http://www.cvedetails.com/ (accessed January, 2016).

[7] G. Díaz and J. R. Bermejo, “Static analysis of source code security: Assessment of tools against samate tests,” *Information and Software Technology*, vol. 55, no. 8, pp. 1462–1476, 2013.

[8] D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis,” *IEEE software*, vol. 19, no. 1, pp. 42–51, 2002.

[9] M. Fang and M. Hafiz, “Discovering buffer overflow vulnerabilities in the wild: an empirical study,” in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2014)*, 2014, p. 23.

[10] “Fortify homepage,” http://www8.hp.com/us/en/software-solutions/application-security/index.html (accessed January, 2016).

[11] “Gartner Group, Gartner Magic Quadrant for Static Application Security Testing,” https://ssl.www8.hp.com/cn/zh/ssl/leadgen/secure_content.html?asset=2053656&module=1823975&siebelid=22908&sectionid=gel (accessed January, 2016).

[12] E. Haugh and M. Bishop, “Testing c programs for buffer overflow vulnerabilities.” in *Network and Distributed System Security Symposium (NDSS)*, 2003.

[13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, 2013, pp. 672–681.

[14] R. W. Jones and P. H. Kelly, “Backwards-compatible bounds checking for arrays and pointers in c programs.” in *Automated Analysis-Driven Debugging (AADEBUG)*, vol. 97, 1997, pp. 13–26.

[15] “Klocwork homepage,” http://www.klocwork.com/ (accessed January, 2016).

[16] K. J. Kratkiewicz, “Evaluating static analysis tools for detecting buffer overflows in c code,” Master’s thesis, Harvard University, 2005.

[17] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities.” in *Proceedings of USENIX Security Symposium*, vol. 32, 2001.

[18] W. Le and M. L. Soffa, “Marple: a demand-driven path-sensitive buffer overflow detector,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008)*, 2008, pp. 272–282.

[19] G. C. Necula, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy code,” in *ACM SIGPLAN Notices*, vol. 37, no. 1, 2002, pp. 128–139.

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *2010 IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 513–528.

[21] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis,” in *Information systems security*, 2008, pp. 1–25.

[22] “Splint homepage,” http://www.splint.org (accessed January, 2016).

[23] J. Viega, J. Bloch, Y. Kohno *et al.*, “A static vulnerability scanner for c and c++ code [c],” in *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC 2000)*, 2000, pp. 257–269.

[24] D. Wagner and D. Dean, “Intrusion detection via static analysis,” in *2001 IEEE Symposium on Security and Privacy (S&P)*, 2001, pp. 156–168.

[25] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, “A first step towards automated detection of buffer overrun vulnerabilities.” in *Network and Distributed System Security Symposium (NDSS)*, 2000, pp. 2000–02.

[26] T. Wang, T. Wei, G. Gu, and W. Zou, “Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection,” in *2010 IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 497–512.

[27] T. Wang, T. Wei, Z. Lin, and W. Zou, “Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution.” in *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2009.

[28] J. Wilander and M. Kamkar, “A comparison of publicly available tools for dynamic buffer overflow prevention.” in *Network and Distributed System Security Symposium (NDSS)*, vol. 3, 2003, pp. 149–162.

[29] Y. Xie, A. Chou, and D. Engler, “Archer: using symbolic, path-sensitive analysis to detect memory access errors,” in *ACM SIGSOFT Software Engineering Notes (SEN)*, vol. 28, no. 5, 2003, pp. 327–336.

[30] R.-G. Xu, P. Godefroid, and R. Majumdar, “Testing for buffer overflows with length abstraction,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*, 2008, pp. 27–38.

[31] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 590–604.

[32] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: exposing missing checks in source code for vulnerability discovery,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2013, pp. 499–510.