International Journal of Systems and Software Security and Protection
Volume 9 • Issue 3 • July-September 2018

What Do We Know About Buffer Overflow Detection?

A Survey on Techniques to Detect A Persistent Vulnerability

Marcos Lordello Chaim, School of Arts, Sciences and Humanities, University of Sao Paulo, Sao Paulo, Brazil
Daniel Soares Santos, Institute of Mathematical Sciences and Computing, University of Sao Paulo, São Carlos, Brazil
Daniela Soares Cruzes, SINTEF Digital, Trondheim, Norway

ABSTRACT

Buffer overflow (BO) is a well-known and widely exploited security vulnerability. Despite the extensive body of research, BO is still a threat menacing security-critical applications. The authors present a comprehensive systematic review on techniques intended to detecting BO vulnerabilities before releasing a software to production. They found that most of the studies addresses several vulnerabilities or memory errors, being not specific to BO detection. The authors organized them in seven categories: program analysis, testing, computational intelligence, symbolic execution, models, and code inspection. Program analysis, testing and code inspection techniques are available for use by the practitioner. However, program analysis adoption is hindered by the high number of false alarms; testing is broadly used but in ad hoc manner; and code inspection can be used in practice provided it is added as a task of the software development process. New techniques combining object code analysis with techniques from different categories seem a promising research avenue towards practical BO detection.

KEYWORDS

Buffer Overflow, Code Weakness, Software Development Process, Software Security, Vulnerability Detection

1. INTRODUCTION

Unintended parties have always tried to access sensitive data stored in software systems. Interconnected computers, though, have multiplied exponentially the harm caused by a breach of security. One of the first largely publicized attacks compromising a great number of interconnected computers was caused by the Morris (or the Internet) worm of 1988. It exploited particulars of the then common DEC VAX machines, the 4 BSD operating system, and the figerd service. However, this exploitation could only be perpetrated by taking advantage of a vulnerability known as buffer overflow or overrun (Erlingsson, 2007).

Buffer overflow (BO) occurs in programs written in languages that do not control the boundaries of arrays during run-time, for instance, FORTRAN, C, and C++ (Spafford, 2003; Viega, Bloch, Kohno, & McGraw, 2000). The goal is to overwrite memory positions to introduce malicious data. The malicious data contain an attack payload, i.e., the malicious code itself, and also a pointer to it. The successful attack does not only copy the malicious data into the memory but also alter the program

DOI: 10.4018/IJSSSP.2018070101

Copyright © 2018, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

execution flow to invoke the attack payload. A BO exploitation allows an attacker to take control of the system. The Morris worm was able to replicate itself because it could attack other computer systems from those whose control were surrendered to it (Spafford, 2003). Another outcome is the crashing of the system, which may lead to Denial of Service (DoS) attacks.

Because BO is associated with the very beginning of the Internet, much research effort has been devoted to address it. One solution is the adoption of memory safe languages such as Java and C# (Horstmann & Cornell, 2005; Drayton, Albahari, & Merrill, 2001), among others. These languages provide mechanisms to control the unintended access to memory positions. For instance, Java and C# check the boundaries of an array when a position beyond the limits are accessed. In these cases, an exception is thrown and the program can take an action to handle it. Another approach is to enforce memory checks in programs written in C/C++ or to add extra code to monitor whether an attack has occurred (Nagarakatte, Zhao, Martin, & Zdancewic, 2009; Ding, He, Wu, Miller, & Criswell, 2012).

With those solutions put in place, BO should be a thing of the past. However, BO exploitations are still common. In July 2012, multiple buffer overflow vulnerabilities were found in the firmware used for VeriFone point-of-sale devices. By exploiting these vulnerabilities, hackers were able to control the terminal and log information input by customers, such as PIN numbers and the magnetic stripe data of a bank card (Constantin, 2012). Among the top security risks for mobile applications (app) tallied in 2016 by the OWASP[1], "client code quality issues" ranks position #7 (OWASP, 2017). This risk category encompasses vulnerabilities such as buffer overflow, format string, use of insecure or wrong APIs, and insecure language constructs.

Why is an old security issue so persistent? The reason is because C and C++ are still used. Many web and mobile applications rely on application programming interfaces (API) written in "old" languages as C and C++ (Sadeghi, Bagheri, Garcia, & Malek, 2017). If an API is vulnerable, the whole application is as well. Moreover, these languages are still used to develop new applications (Teixeira et al., 2015). C is the second most used language after Java to develop Internet of Things (IoT) applications, according to a recent survey conducted by the Eclipse IoT Working Group and other organizations (Skerret, 2017). The threat to IoT devices is particularly disturbing because they are supposed to take care of people's intimate activities (Padmanabhuni & Tan, 2015b).

Techniques that provide protection against BO at run-time tend to impose a performance penalty. As result, they might be overlooked when performance is an issue compromising a system. To avoid BO vulnerabilities being neglected, one should detect them during the development of the software. However, the large number and variety of approaches and techniques that have been proposed to deal with buffer overflow vulnerabilities make hard to obtain a good understanding of existing solutions.

Studies such as Erlingsson (2007); Lhee & Chapin (2003); Wilander and Kamkar (2003); Younan, Joosen, & Piessens (2012) provide an overview of dynamic and run-time countermeasures to protect against BO attacks. On the other hand, there are still few studies describing the state-of-the-art, or conducting experimental evaluation of techniques, to detect BO during software development (Pozza, Sisto, Durante, & Valenzano, 2006; Ye, Zhang, Wang, & Li, 2016). In addition, there is evidence that these approaches have being underutilized by security professionals nowadays. Fang and Hafiz (2014) performed an empirical study on 58 reporters of BO vulnerabilities of the securityFocus repository and concluded that almost none of them are in fact used in this context. According to Johnson, Song, Murphy-Hill, & Bowdidge (2013), false positives and developer overload, among other issues, actually may be preventing the utilization of these techniques in practice. Despite that, detection techniques are considered more systematic and scalable, when compared to protection techniques (Pozza, Sisto, Durante, & Valenzano, 2006; Ye, Zhang, Wang, & Li, 2016).

The evidence of the underutilization of BO detection approaches and the lack of studies describing the state-of-the-art motivate the conduction of more comprehensive literature reviews, surveys, comparative analysis, and empirical studies, which are essential to allow a greater recognition and a broader use of these solutions by the industry.

In this context, we present a systematic review on techniques and tools to detect BO vulnerabilities. This study aims at identifying (1) the relevant BO vulnerabilities, (2) how BO detection has evolved, (3) the main techniques to detect BO during the development phase, and (4) which techniques can be used at industrial settings. Our goal is to describe the state of the practice and the art on techniques to tackle BO vulnerabilities in the development phase. We conclude by presenting recommendations to practitioners and researchers interested in BO detection.

The remaining of the paper is organized as follows. In the following section, we describe the procedure utilized to conduct this systematic literature review. The results are presented next; first the relevant BO vulnerabilities are described, then the techniques identified to tackle them in terms of goals, source of analysis, and categories. The discussion section follows guided by the research questions. The paper finishes with the conclusions.

2. SYSTEMATIC LITERATURE REVIEW — SLR

A Systematic Literature Review (SLR) is an approach for finding and summarizing available evidence on a particular research topic. Our SLR was performed following the process proposed by Kitchenham & Charters (2007), which define three main phases: (1) Planning: in this phase, the SLR protocol is defined, which refers to a pre-determined plan that describes the research objectives and how the SLR will be conducted in order to achieve such objectives; (2) Conduction: during this phase, primary studies are identified, selected, and evaluated according to the SLR protocol defined in the planning phase; (3) Reporting: in this phase, a final report is elaborated and presented. The next sections present these three phases in details.

2.1. Planning Phase

The objective of this SLR is to identify, analyze, and synthesize the state-of-the-art on buffer overflow vulnerabilities and current initiatives addressed to detect them. According to this purpose, the following research questions (RQ) were established:

R1: Which are the relevant vulnerabilities related to buffer overflow?
R2: How has buffer overflow detection evolved?
R3: Which are the techniques available to detect buffer overflow vulnerabilities before the release of the software?
R4: Which are the techniques scalable to be utilized at industrial settings?

To find the main primary studies to answer the aforementioned RQs, two searching strategies were considered: (1) The searching in a representative set of publication databases, such as: Scopus[2]; and Web of Science[3], that are comprehensive research platforms considered the largest databases of scientific studies. According to the best of our knowledge, these databases are effective and representative to conduct systematic reviews in the Software Engineering context. Furthermore, (2) snowballing from reference lists of the identified articles. This practice was used to identify additional relevant studies that, for some reason, were not found in the search on publication databases.

The search string used on publication databases was composed from the main keywords "buffer overflow" and its synonym "buffer overrun". The final search strings were tailored according the specification of each publication database.[4]

The identification of relevant primary studies was supported by a set of Inclusion Criteria (IC) and Exclusion Criteria (EC). These criteria were used to help the selection of those studies that are relevant to answer the research questions and exclude those studies that are not. In this sense, the defined IC and EC were:

Inclusion Criteria:

IC1: The purpose of the study are techniques to detect buffer overflow vulnerabilities

Exclusion Criteria:

EC1: The study does not focus on buffer overflow vulnerabilities
EC2: The study does not propose techniques to detect buffer overflow vulnerabilities
EC3: The study is an editorial, keynote, opinion, tutorial, poster or panel
EC4: The study is a previous version of a more complete study about the same research
EC5: The paper language is different from English
EC6: The paper is duplicated
EC7: The full paper is not available

As can be perceived, only primary studies written in English were considered in this SLR, since most of research in Computer Science has been reported in this language.

After the searching on the publication databases, the process to select the primary studies was divided into four steps: (1) Selection based on paper's title and abstract. As result, a set of primary studies possibly related to the research topic is obtained. The introduction and the conclusion sections of each primary study might also be considered when necessary; (2) Selection based on paper's full text. In this step, the set of relevant primary studies are identified in order to answer the research questions; (3) Selection based on paper's related works (snowballing). This additional step might be a great source of evidence, since an included study often presents related works in the same research area. The selection of all relevant studies is guided by the application of inclusion and exclusion criteria; and (4) Data Extraction and Synthesis. The relevant information from each selected study is extracted and synthesized in order to answer the research questions. Disagreement among the researchers in any of the previous steps can be solved by consensus meetings.

2.2. Conduction Phase

This SLR was conducted by two software engineering researchers and one PhD student. The work started in 2014 January and was finalized in 2017 May. In this section, we describe in details the procedures for selecting the studies.

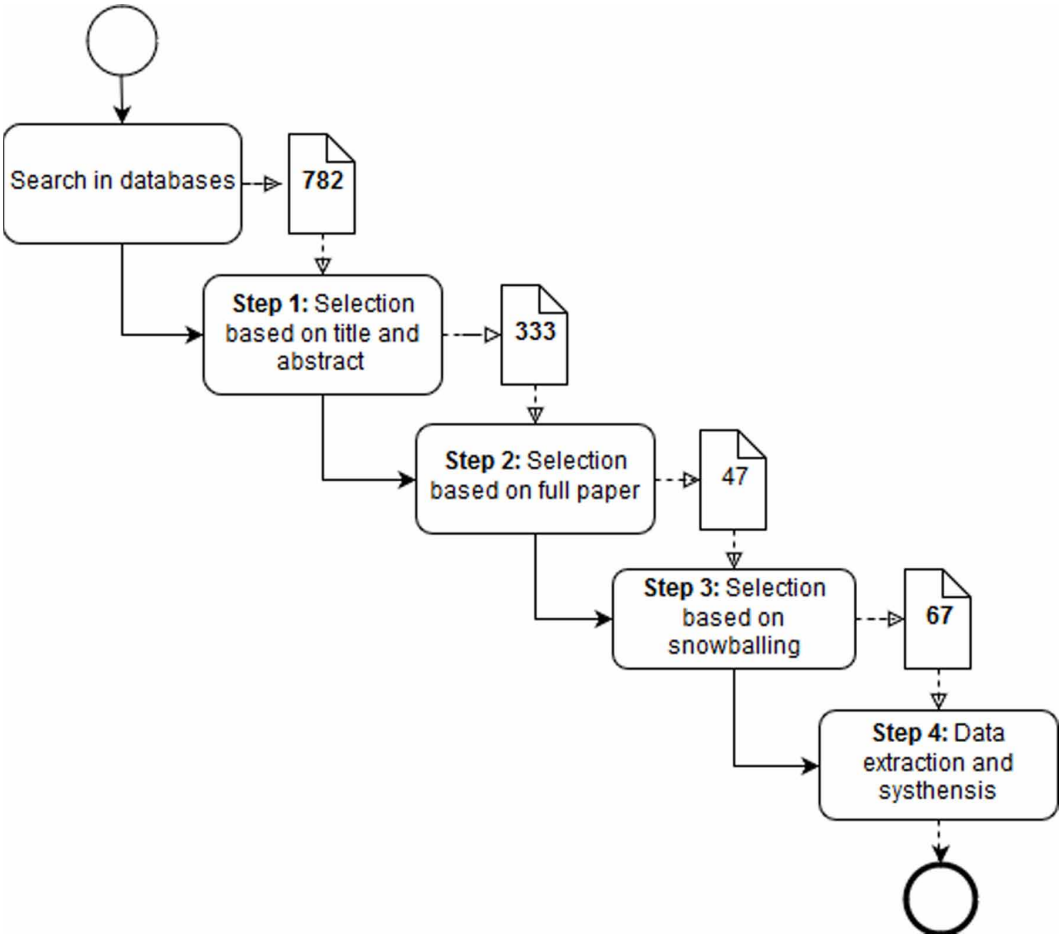
According to the planning previously established the primary studies were identified on the selected databases. However, the searching resulted in overlapping studies, which required the exclusion of duplicates through a manual filtering. In total, 782 different primary studies were identified. All these studies were analyzed reading their title and abstract and by the application of the inclusion and exclusion criteria to identify their relevance. After that, we selected 333 primary studies, which were read in full. From that, 47 studies were included in our SLR. During the full reading of the included studies, related works were identified and selected through the snowballing. Finally, a total of 67 studies were included to our SLR and submitted to the extraction of relevant information to answer the research questions. Figure 1 summarizes the process of selection of primary studies.

2.2.1. Threats to Validity

The main threats to validity identified in this SLR are described as follows:

- **Missing of Important Primary Studies:** the search for studies related to buffer overflow was conducted in the most relevant publication databases. Moreover, no limit was placed on the date of publications. During the search, conference papers, journals, and technical reports were considered. In spite of the effort to include all relevant evidence in this research, it is possible that primary studies were missed;

Figure 1. Process of selection of primary studies



- **Reviewers Reliability:** the authors of this work are not aware of any bias that may have been introduced during the analysis process. However, it might be possible that the conclusion about the studies evaluated have been influenced by the opinion of the reviewers; and
- **Data Extraction:** another threat refers to how the data were extracted from the primary studies. Not all extracted information was obvious to answer the research questions and some data had to be interpreted. Furthermore, in case of disagreement among the reviewers, a discussion was conducted to ensure that a full agreement was reached.

In particular, we have dedicated special effort to completely cover this research area as impartially as possible.

3. RESULTS

In what follows, we report the main results of our SLR. Firstly, the main buffer overflow vulnerabilities are reported. They serve the purpose of describing how BO vulnerabilities are exploited by attackers and also how the techniques cope with them. The techniques in the selected studies are then presented. The presentation is organized by goals, object of analysis (source or object code), and categories. Six

Figure 2. C Program example 1

```
1  void main(int argc, char *[] argv) {
2  ...
3  func1(4,5);
4  ...
5  }
6
7  void func1(int a, int b){
8  char name[100];
9
10 ...
11 /* Point 1 */
12
13 scanf("%s",name);
14
15 /* Point 2 */
16 ...
17 }
```

categories were identified, namely, code inspection, computational intelligence, models, program analysis, symbolic execution and testing. Techniques can often be classified in more than one category; in such cases, they are described in their main category, and their relationship with other categories are reported using Venn's diagrams and tables.

3.1. Relevant Buffer Overflow Vulnerabilities

Buffer overflow or overrun (BO) is a program vulnerability allowing attackers to overwrite particular memory locations. By doing so, they are able to execute malicious code. It is associated with programs written in C or C++ because these languages do not protect the memory from being overwritten or accessed during the execution of a program. BO exploits can be achieved by directly overwriting the memory (stack or heap) or indirectly by causing an integer overflow.

Most of the papers addresses memory errors in general (see Section *Discussion*). Few studies pinpoint which vulnerability they address. Nevertheless, specific BO vulnerabilities were mentioned. They are explained below to help the reader understand the techniques we identified. We start off presenting the typical BO exploit, called stack-based BO, then we discuss other BO exploits.

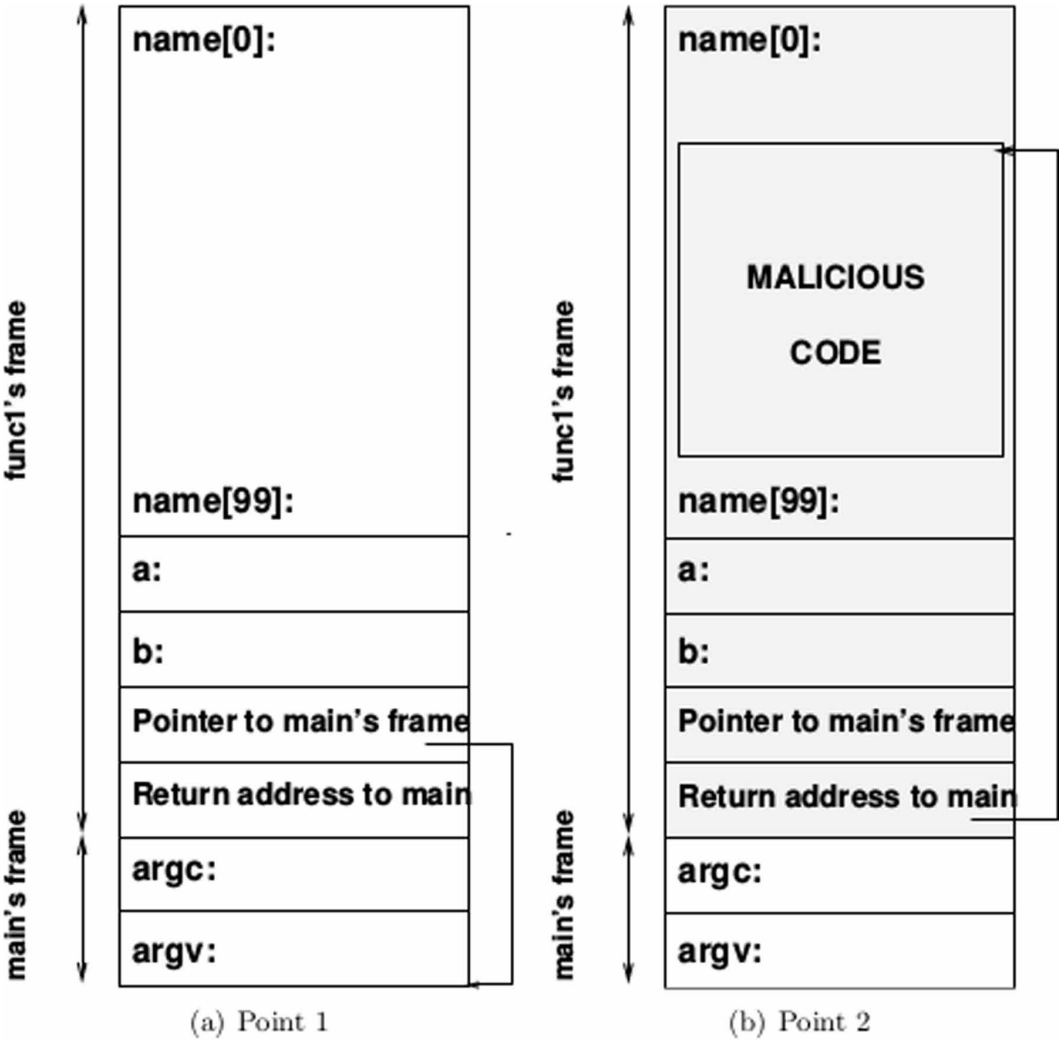
3.1.1. Stack-Based Buffer Overflow

Many program languages, like C and C++, utilize dynamic memory stacks to implement functions[5] and recursion (Sebesta, 2012). In general, they are organized in activation records or stack frames to store fixed size data such as parameters, local variables, arrays and addresses. Figure 2 shows an skeletal C program (example 1) used to illustrate a memory stack.

Figure 3 presents the stack frames of functions main() and func1() during the execution. At *Point 1* of Figure 2, shown in Figure 3(a), main() and func1() frames are on the memory stack. main()'s frame contains only local variables since it is where the program execution starts. However, func1()'s frame contains the return address to resume the caller's execution (in this case, main()), a pointer to the caller's frame, and the local variables — parameters a and b and the array name.

In many computer architectures (e.g., the x86 family), the stack grows down meaning that the callee's frame (in our case, func1()) will be located at lower memory addresses than the caller's

Figure 3. Stack memory for program example 1



frame (Younan, Joosen, & Piessens, 2012). In Figure 3, name[0] has a lower address than name[99] emulating one of these architectures. At line 13, the scanf("%s",name) command reads a string from the keyboard and insert it in name starting from name[0].

Because programs written in C do not control the boundaries of arrays, an attacker, knowledgeable of the vulnerability present in the example program, can craft an input string such that the return address to main() will be overwritten after the execution of scanf("%s",name). Figure 3(b) shows the memory stack at *Point 2* (line 15 of Figure 2), after a successful attack. The light gray areas indicate the portions of the memory stack compromised by the attack.

The attacker provided a string whose size was bigger than the size of name enough to overwrite variables a and b, the pointer to main()'s frame and the return address. The string was crafted in such a way that the value overwritten to the return address points to an address inside name. It turns out the string inputted by the attacker contains a malicious code that starts at that memory address. As a result, when func1() ends, the execution does not return to main(), but to the malicious code.

There are variants of this attack that corrupts the pointer to the caller's frame or exception-handler pointers present in the stack (Erlingsson, 2007; Yan, Liu, & Meng, 2015). Typically, the attack payload launches a "shell" command interpreter under their control (Erlingsson, 2007).

3.1.2. Heap-Based Buffer Overflow

Heap-based BO exploitations are similar to stack-based exploitations since they overwrite an array and an adjacent pointer to a sub-routine. The difference is that it is not the pointer to the caller function that is overwritten, but to a function that might be called by other functions. Consider the simplified C program (example 2) presented in Figure 4.

Below *Point 1*, between lines 2 to 5, there is a record definition. This record is composed of a pointer to an array of integers myInts (line 3) and a pointer to a function sort (line 4) to supposedly sort the array of integers. Line 5 defines the name of the record type — SortableInts.

Lines 8 and 9, after Point 2, contain the definition of two sorting functions that might be addressed by pointer sort. The code after Point 3 describes the creation of heap objects. Lines 16 and 17 show the memory allocation for an instance (object) of SortableInts. Line 18 reads from the keyboard the size of the array of integers. At Line 19, the array of integers is created. The memory allocated to it is calculated by multiplying the size inputted by the user (nInts) and the number of bytes to store an integer given by sizeof(int).

Lines 22 to 30, after *Point 4*, define the contents of the heap objects. From line 22 to line 26, the sorting function is selected and assigned to the instance of SortableInts. The values of the array are initialized from lines 27 to 29. At *Point 5*, line 33, the chosen sorting function is called. Had this example program been written in C++, SortableInts would be a class and sort a virtual function.

Let us say an attacker manages to make the size of myInts smaller than originally expected. Then, the for at line 27 will be vulnerable to malicious data being written beyond the limits of myInts. If the attacker is successful in addressing function pointer sort to a malicious code, the system will be compromised. In the next section, we show how she or he can achieve such a goal.

3.1.3. Integer Buffer Overflow

Integer overflow can be used as a precursor of a BO exploitation. It is used to make a buffer smaller than expected so that malicious data can be copied into sensitive memory positions. If an integer overflow occurs when calculating the size of a buffer, less memory might be allocated to it. As a result, the buffer can be exploited.

Consider a 32-bit unsigned integer variable[6]. This variable stores values from zero to 2^{32} - 1. When such a variable receives 2^{32}, it needs an extra bit for the most significant bit (1) to represent 2^{32}. Since there is not an extra bit, the new value is 0 because the remaining bits are all 0.

We go back to example 2 (Figure 4) to illustrate how integer overflows are used to achieve BO exploitations. The size of myInts is inputted by the user at line 18. A 32-unsigned int variable, nInts, declared in line 12, stores it. The memory allocated for myInts is calculated by multiplying nInts by four—the size of an integer in bytes given by sizeof(int). The goal of the attacker is to make this multiplication overflow. By doing so, she or he will have the value of nInts much greater that the memory allocated to myInts, which grants the exploitation of the for command (line 27) to copy malicious data into sensitive memory positions.

So, the value inputted to nInts is pivotal. It will be multiplied by four (2^2); the obvious input value to make the multiplication overflow is 2^{30}. The result 2^{32} does overflow, but the final value is zero, which causes no memory to be allocated to myInts. Hardly, this situation would go undetected because a common C programming practice is to test whether a pointer is different of NULL (zero) after any memory allocation. Figure 4 does not show these tests to reduce the size of the example program.

To achieve a small buffer and a high value for nInts, the attacker should input $2^{30} + 1$, which multiplied by four will give $2^{32}+4$. This result overflows and the final value is 4. Thus, myInts will store a single integer, but the value of nInts, which controls the for command, will be much bigger—$2^{30} + 1$. Thus, the door is open to the introduction of malicious data.

Figure 4. C program example 2

```
1  /* Point 1: Type definition */
2  typedef struct sortableints {
3                  int * myInts;
4                  void (*sort)(int *);
5  } SortableInts;
6
7  /* Point 2 : Sorting functions */
8  void quick(int * v) {...}
9  void buble(int * v) {...}
10
11 void main(int argc, char *argv[]) {
12 unsigned int nInts;
13 int i=0, val = 0; char alg;
14
15 /* Point 3: Heap objects creation */
16 SortableInts * m = (SortableInts *)
17                  malloc (sizeof(SortableInts));
18 nInts = atoi(argv[1]);
19 m->myInts = (int *) malloc(nInts * sizeof(int));
20
21 /* Point 4: data structure initialization */
22 scanf("%c",&alg);
23 switch(alg) {
24     case 'b': m->sort = &buble; break;
25     default: m->sort = &quick;
26 }
27 for(i=0; i < nInts; ++i) {
28     scanf("%d", &val);
29         m->myInts[i] = val;
30 }
31
32 /* Point 5: Sorting function call*/
33 m->sort(m->myInts);
34 }
```

3.2. Techniques to Detect Buffer Overflow Vulnerabilities

Next, we present the most relevant techniques identified by the SLR. They are described taking into account their goals, source of analysis, and categories.

3.2.1. Goals and Source of Analysis

Techniques to detect buffer overflow vulnerabilities have different goals, namely, to report excerpts of code vulnerable to BO exploits, to crash the program to provide evidence of possible exploitations, and to repair vulnerable locations. Crashing and reporting techniques are present since the inception of BO detection techniques. On the other hand, techniques that repair code have appeared recently.

Code reports list lines of code to be inspected by the developers to check the existence of BO vulnerabilities. The main issues concerning code reports are the existence of false positives and false negatives which hinders the adoption of the techniques in practice. Crashing the system is an indication that a BO attack may occur; however, additional conditions should hold to an attack go through: the buffer should be big enough to contain malicious code and it also should reach the return address in the stack. Thus, crashing requires that the exploitation be confirmed and the location of the vulnerability determined. Repairing, in turn, can only occur after the vulnerability in the code is located. As a result, it utilizes the results of reporting techniques.

Figure 5 contains a Venn's diagram which associates the goals (reporting, crashing, and repairing) to the techniques. The intersections between the techniques indicate they have both goals or the main goal is achieved by first achieving the secondary goal. Reporting is the most common goal. There are techniques that combine reporting and crashing goals. The main goal may vary in these studies. For some of them, possible vulnerable positions are first identified then test cases are derived to crash the system and to confirm an exploitation (Ghosh, O'Connor, & McGraw, 1998; Padmanabhuni & Tan, 2015b). The other way around is also possible: from the crashing of the system, the vulnerable location is reported (Gupta, He, Zhang, & Gupta, 2005; Jeffrey, Gupta, & Gupta, 2008; Padaryan, Kaushan, & Fedotov, 2015). Additionally, there are approaches that first run a test case and then identify constraints associated with vulnerabilities (Li & Shieh, 2011; Mouzarani, Sadeghiyan, & Zolfaghari, 2015). Few techniques (Bruschi, Rosti, & Banfi, 1998; Chen, et al., 2013; Duraes & Madeira, 2005; Grosso, Antoniol, Merlo, & Galinier, 2008; Shahriar & Zulkernine, 2008; Woodraska, Sanford, & Xu, 2011) aim only at crashing the program. All repairing techniques (Gao, Wang, & Li, 2016; Muntean, Kommanapalli, Ibing, & Eckert, 2015; Novark, Berger, & Zorn, 2007; Zhang, Wang, Wei, Chen, & Zou, 2010) also report vulnerabilities.

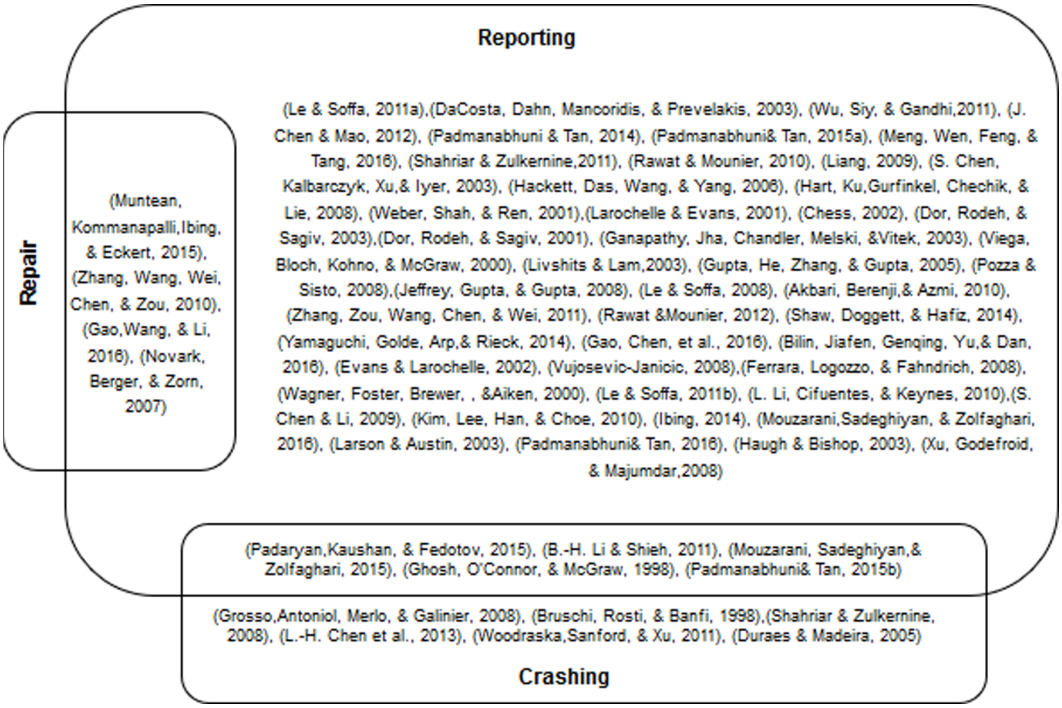
Most of the studies utilizes as their source of analysis programs written C or C++. Only six studies (Bruschi, Rosti, & Banfi, 1998; Chen, et al., 2013; Duraes & Madeira, 2005; Ferrara, Logozzo, & Fähndrich, 2008; Padmanabhuni & Tan, 2015a; Padmanabhuni & Tan, 2015b) analyze object code; in general, x86 code is analyzed, excepting one study (Ferrara, Logozzo, & Fähndrich, 2008) which utilizes MSIL – Microsoft Intermediate language – as its source of analysis.

3.2.2. Categories of BO Detection

For each selected paper, we identified the main technique supporting the study and classified it into a category. The following categories to implement BO detection were identified: code inspection, computational intelligence, models, program analysis, symbolic execution and testing. In many papers, a combination of techniques from different categories is utilized. Figure 6 presents these categories and their relationships in a Venn's diagram; Table 1 contains a similar information in a textual form. The column Qty. indicate the number of paper that focus only on a category. Additionally, column Total indicate the total number of papers that directly or indirectly address that category.

Program analysis (PA) is the largest category, as shown by its size in Figure 6, with 45 studies using some form of program analysis and 17 using it as a solo technique. Additionally, all other categories, excepting empirical studies, have some study that utilize program analysis to achieve its goal. Models (MD) and testing (Tst) are frequent among the studies, being used in 17 studies. Symbolic execution (SE) is present in 12 studies. Generally, models and symbolic execution are used in combination with techniques from other categories; only three studies (Chen, Kalbarczyk, Xu, & Iyer, 2003; Hackett, Das, Wang, & Yang, 2006; Hart, Ku, Gurfinkel, Chechik, & Lie, 2008) utilize models and one (Padaryan, Kaushan, & Fedotov, 2015) symbolic execution without the support of other techniques. Computational Intelligence (CI) is used in seven studies; most of them in combination with program analysis techniques. Code inspection (CoI) is a category of only three papers. Finally, empirical studies (ES), in general, comprise comparison of tools. As the tools compared utilize different approaches, we decided to create a category of its own comprising six studies. The exception was a study (Chen, Dean, & Wagner, 2004) that utilizes a single tool based on models in the empirical study.

Figure 5. Techniques' goals



In the next sections, we describe the techniques utilized to detect BO vulnerabilities organized by the identified categories. We start off with program analysis which is the prevalent category.

3.2.3. Program Analysis

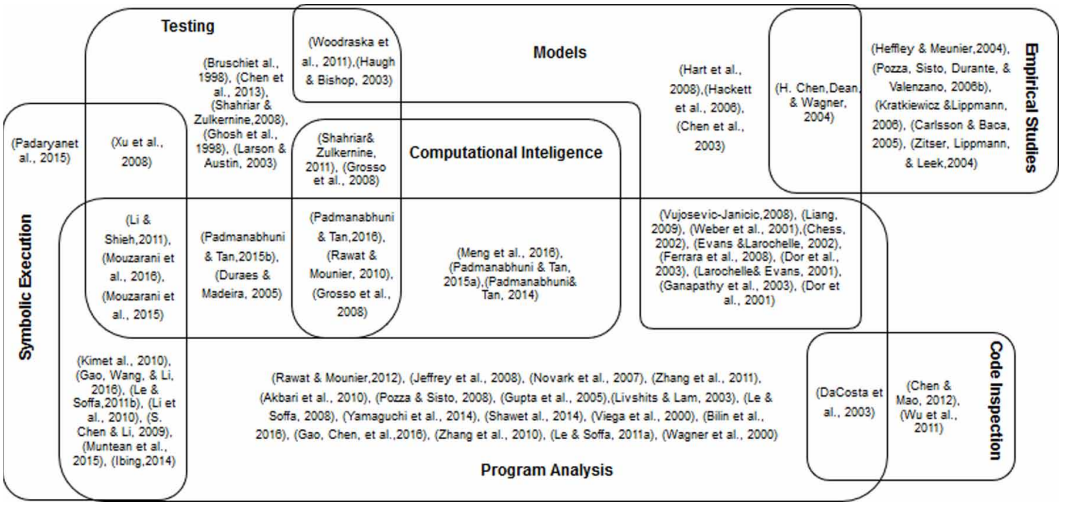
Program analysis techniques aim to analyze source code to support activities like testing, debugging, and maintenance (Binkley, 2007). Program is in this context a broad term. It encompasses specifications, models, diagrams, and source and object code (Harman, 2010; Jackson & Damon, 2000). Program analysis is the most used approach for BO detection because its techniques manipulate (source or object) code either to report vulnerabilities or to support the application of other approaches.

Program analysis techniques can be static or dynamic. Static analysis obtains information from the program that is valid for all possible executions. Dynamic analysis instruments the program to collect information as it runs. The results of a dynamic analysis are valid only for a particular execution (Jackson & Damon, 2000). A sound analysis is valid for all the runs of the program whereas an unsound analysis makes no guarantees, but often can quick produce correct or approximate results (Binkley, 2007). Sound analysis comes at a price of less precise information, implying more false positives.

Flow-sensitive analysis takes into account the execution order of the statements whilst flow-insensitive analysis is oblivious to the flow of execution. Context-insensitive is the analysis for which the result is valid for all contexts in which a procedure is called. Context-sensitive, in turn, refers to analyses that produce a different result for each different calling context (Binkley, 2007; Jackson & Damon, 2000). Flow- and context-sensitive algorithms tend to be more precise but also costlier; flow- and context-insensitive ones are less precise but tend to be scalable for large applications.

Several program analysis techniques are useful for BO detection. Compiling techniques such as syntactic and semantic analysis of the code support scanning of patterns and program transformation. Typically, they build abstract trees and collect semantic data (e.g., type and size) of variables. Static analysis techniques – data-flow analysis and abstract interpretation (Aho, Lam, Sethi, & Ullman,

Figure 6. Categories of techniques for BO detection



2007; Nielson, Nielson, & Hankin, 1999) – determine facts at particular points of the program that are valid for all inputs. The reaching definitions, i.e., the assignments of values to variables that reach a particular program point, are examples of facts. Static analysis iterates on graph representations of the program to determine these facts. Compiling techniques can be seen as lightweight static analysis while data-flow analysis and abstract interpretation as heavyweight static analysis.

Taint analysis is a particular case of reaching definitions that determines if a possible tainted input (source) reaches a particular vulnerable location (sink) of the program (e.g., a strcpy() function call). Points-to or pointer analysis, program slicing and delta debugging are also useful for BO detection. Points-to analysis determines the locations (variables or memory) a pointer is referring to (Binkley, 2007). Program slicing (Weiser, 1984) selects statements of the program, called slice, that impact the value of a set of variables at a particular point of the program. Delta Debugging (Zeller, 2009) determines causes of failures that occur during execution, using the difference between one passing and one failing run. These differences are reduced to obtain a minimal set that causes the failure. Finally, memory tracking records the state of the memory in order to mine patterns associated with buffer overflow and other memory errors (e.g., dangling pointers).

Table 2 describes the program analysis techniques used in the selected studies. Static analysis with 20 studies is the most utilized technique. Moreover, taint and points-to analysis and program slicing are based on static analysis algorithms; thus, heavyweight analysis is prevalent among the program analysis techniques. Nevertheless, lightweight techniques (compiling techniques and program transformations) are not far behind, being utilized by 19 studies. Table 2 also presents the techniques in terms of context and sensitivity to the flow of execution. We report these data as they were mentioned in the studies. Since flow- and context-insensitive techniques are more scalable, the number of studies using these approaches should be greater. Few studies report the insensitiveness to context and to flow of the execution, likely because they are the standard approaches. Purely dynamic program analysis techniques as delta debugging and memory tracking are yet scarcely used. Next, the studies whose main technique is based on program analysis are discussed. Those studies for which program analysis is subsidiary are only referred to and discussed in the following sections.

Many works utilize compiling techniques to support BO detection (Bilin, Jiafen, Genqing, Yu, & Dan, 2016; Duraes & Madeira, 2005; Evans & Larochelle, 2002; Meng, Wen, Feng, & Tang, 2016; Padmanabhuni & Tan, 2016; Padmanabhuni & Tan, 2014; Yamaguchi, Golde, Arp, & Rieck, 2014; Zhang, Zou, Wang, Chen, & Wei, 2011), there are works that rely only on compiling techniques to

detect BO vulnerabilities. Akbari, Berenji, & Azmi (2010) first traverse the parse tree and annotate it, then vulnerability detection rules are performed on the annotated tree to find code weaknesses. ITS4 (Viega, Bloch, Kohno, & McGraw, 2000) takes one or more C or C++ source files as input and breaks each into a stream of tokens. After scanning a file, ITS4 examines the resultant token stream, comparing identifiers against a database of "suspects." Rawat & Mounier (2012) define a vulnerability pattern called Buffer Overflow Inducing Loops (BOIL). A loop is a BOIL if there is memory write within the loop and that memory is changing within the loop. They analyze the binary executable of the application to find BOP functions (i.e. functions with BOILs). Splint (Evans & Larochelle, 2002) uses lightweight static analysis to detect likely vulnerabilities in programs, being able to detect a wide range of implementation flaws by exploiting annotations added to programs. Yamaguchi, Golde, Arp, & Rieck, (2014) introduce a representation of source code called code property graph that merges abstract syntax trees, control flow graphs and program dependence graphs, into a joint data structure. Common vulnerabilities, e.g., stack and integer BO, are modeled so that queries can be made by traversing the code property graph.

Program transformation is in general used as a subsidiary technique (Chen & Li, 2009; Chess, 2002; Dor, Rodeh, & Sagiv, 2003; Dor, Rodeh, & Sagiv, 2001; Muntean, Kommanapalli, Ibing, & Eckert, 2015; Vujosevic-Janicic, 2008). We single it out from compiling techniques because a transformed program is not an intermediary result (e.g., an abstract tree). It converts a program representation, in general, source code, into another representation amenable to the application of a particular technique. The result can be a source-to-source semantic-preserving transformation of the program (Dor, Rodeh, & Sagiv, 2003; Dor, Rodeh, & Sagiv, 2001), a set of conditions (Chess, 2002; Vujosevic-Janicic, 2008), code patterns to fix BO vulnerabilities (Muntean, Kommanapalli, Ibing, & Eckert, 2015) or a simplified version of the program (Chen & Li, 2009). Program transformation is the main technique when it fixes BO vulnerabilities. Shaw, Doggett, & Hafiz (2014) introduce two

Table 1. Categories of techniques in the studies

Category	Studies	Qty.	Total
Program analysis	(Zhang et al., 2011; Le & Soffa, 2011b; Li et al., 2010), (Akbari et al., 2010; Liang, 2009; S. Chen & Li, 2009; Pozza & Sisto, 2008) (Gupta et al., 2005; Mouzarani et al., 2015), (Muntean et al., 2015; Padmanabhuni & Tan, 2015a; Padman-abhuni & Tan, 2015b; Zhang et al., 2010; Li & Shieh, 2011; Kim et al., 2010; Novark et al., 2007; Vujosevic-Janicic, 2008; Wagner et al., 2000; Ganapathy et al., 2003; Dor et al., 2001; Gao, Wang, & Li, 2016; Meng et al., 2016; Mouzarani et al., 2016; Bilin et al., 2016; Rawat & Mounier, 2010; Fer-rara et al., 2008; Le & Soffa, 2008; Duraes & Madeira, 2005), (Dor et al., 2003; Larochelle & Evans, 2001; Weber et al., 2001; Chess, 2002; Evans & Larochelle, 2002; Viega et al., 2000; Ibing, 2014; Shaw et al., 2014; Padmanabhuni & Tan, 2014; Jeffrey et al., 2008; Yamaguchi et al., 2014; Gao, Chen, et al., 2016; Padmanabhuni & Tan, 2016; DaCosta et al., 2003; Le & Soffa, 2011a; Livshits & Lam, 2003; Rawat & Mounier, 2012)	17	45
Computational Intelligence	(Shahriar & Zulkernine, 2011; Meng et al., 2016; Grosso et al., 2008; Padmanabhuni & Tan, 2015a; Rawat & Mounier, 2010), (Padmanabhuni & Tan, 2016; Padmanabhuni & Tan, 2014)	0	7
Symbolic Execution	(Li & Shieh, 2011; Kim et al., 2010; Gao, Wang, & Li, 2016; Mouzarani et al., 2016; Le & Soffa, 2011b; Li et al., 2010; Chen & Li, 2009; Xu et al., 2008; Mouzarani et al., 2015; Muntean et al., 2015; Padaryan et al., 2015; Ibing, 2014)	1	12
Testing	(Li & Shieh, 2011; Woodraska et al., 2011; Shahriar & Zulkernine, 2011; Bruschi et al., 1998; Mouzarani et al., 2016; Chen et al., 2013; Grosso et al., 2008; Shahriar & Zulkernine, 2008; Xu et al., 2008; Mouzarani et al., 2015; Pad-manabhuni & Tan, 2015b; Rawat & Mounier, 2010; Duraes & Madeira, 2005; Ghosh et al., 1998; Larson & Austin, 2003; Haugh & Bishop, 2003; Padmanabhuni & Tan, 2016)	5	17
Models	(Woodraska et al., 2011; Vujosevic-Janicic, 2008; Hart et al., 2008; Hackett et al., 2006; Chen et al., 2003; Liang, 2009), (Ferrara et al., 2008; Dor et al., 2003; Larochelle & Evans, 2001; Weber et al., 2001; Chess, 2002; Evans & Larochelle, 2002; Chen et al., 2004; Haugh & Bishop, 2003; Grosso et al., 2008; Ganapathy et al., 2003; Dor et al., 2001)	3	17
Code Inspection	(Chen & Mao, 2012; Wu et al., 2011; DaCosta et al., 2003)	2	3
Empirical studies	(Heffley & Meunier, 2004; Pozza et al., 2006b; Kratkiewicz & Lippmann, 2006; Carlsson & Baca, 2005; Zitser et al., 2004; Chen et al., 2004)	5	6

Table 2. Program analysis techniques used in the studies

Technique	Studies	Qty
Compiling	(Viega et al., 2000; Evans & Larochelle, 2002; Duraes & Madeira, 2005; Akbari et al., 2010; Zhang et al., 2011; Rawat & Mounier, 2012; Yamaguchi et al., 2014; Padmanabhuni & Tan, 2014; Bilin et al., 2016; Padmanabhuni & Tan, 2016; Meng et al., 2016; DaCosta et al., 2003)	12
Program transformation	(Dor et al., 2001; Chess, 2002; Dor et al., 2003; Vujosevic-Janicic, 2008; Chen & Li, 2009; Shaw et al., 2014; Muntean et al., 2015)	7
Static analysis	(Wagner et al., 2000; Weber et al., 2001; Evans & Larochelle, 2002; Larochelle & Evans, 2001; Livshits & Lam, 2003; Vujosevic-Janicic, 2008; Ferrara et al., 2008; Pozza & Sisto, 2008; Le & Soffa, 2008; L. Li et al., 2010; Kim et al., 2010; Zhang et al., 2010; Li & Shieh, 2011; Le & Soffa, 2011b; Ibing, 2014; Padmanabhuni & Tan, 2015a; Bilin et al., 2016; Gao, Wang, & Li, 2016; Gao, Chen, et al., 2016; Le & Soffa, 2011a)	20
Taint analysis	(Ganapathy et al., 2003; Livshits & Lam, 2003; Dor et al., 2003)	3
Slicing	(Ganapathy et al., 2003; Gupta et al., 2005; Rawat & Mounier, 2010; Zhang et al., 2010; Zhang et al., 2011)	5
Point-to analysis	(Ganapathy et al., 2003; Livshits & Lam, 2003; Dor et al., 2003)	3
Context-sensitive	(Weber et al., 2001; Livshits & Lam, 2003)	2
Context-insensitive	(Livshits & Lam, 2003)	1
Flow-sensitive	(Weber et al., 2001; Livshits & Lam, 2003; Vujosevic-Janicic, 2008; Le & Soffa, 2008; L. Li et al., 2010; Le & Soffa, 2011b; Le & Soffa, 2011a)	7
Flow-insensitive	(Livshits & Lam, 2003)	1
Delta debugging	(Gupta et al., 2005)	1
Memory tracking	(Novark et al., 2007)	1

transformations that fix buffer overflows in C programs by prescribing two frequently used security solutions.

Static analysis determines facts at program points that enable BO detection (Evans & Larochelle, 2002; Ferrara, Logozzo, & Fähndrich, 2008; Gao, Wang, & Li, 2016; Ibing, 2014; Kim, Lee, Han, & Choe, 2010; Larochelle & Evans, 2001; Li & Shieh, 2011; Li, Cifuentes, & Keynes, 2010; Livshits & Lam, 2003); (Padmanabhuni & Tan, 2015a; Vujosevic-Janicic, 2008). Nevertheless, different facts can be statically determined for BO detection. Wagner, Foster, Brewer, & Aiken (2000) formulate the BO detection problem as an integer constraint or range propagation problem (it establishes which values can be bound to an integer variable) and utilize static analysis to solve it. Pozza & Sisto (2008) combine taint analysis and value range propagation in the gcc compiler. Mjolnir (Weber, Shah, & Ren, 2001) uses a fixed-point algorithm to solve safety constraints. Marple (Le & Soffa, 2008) checks whether each buffer access in the program is safe by propagating backwards value range information along the control flow towards the entry. Only statements that can reach the buffer access need to be examined to determine the vulnerability. In this sense, it utilizes a flow-sensitive and demand-driven approach.

Points-to (Ganapathy, Jha, Chandler, Melski, & Vitek, 2003) taint analysis (Ganapathy, Jha, Chandler, Melski, & Vitek, 2003; Liang, 2009; Mouzarani, Sadeghiyan, & Zolfaghari, 2016; Mouzarani, Sadeghiyan, & Zolfaghari, 2015; Padmanabhuni & Tan, 2015b) and program slicing (Ganapathy, Jha, Chandler, Melski, & Vitek, 2003; Rawat & Mounier, 2010) utilize static analysis

algorithms to establish facts (e.g., variables or memory referred to, slices, tainted variables) at particular points of the program. Livshits & Lam (2003) use a hybrid approach to achieve efficient and precise points-to analysis to detect BO and format string vulnerabilities. Pointers accessed through simple access paths are analyzed precisely; an efficient flow- and context-insensitive analysis is used for other indirect accesses to reduce the costs of the analysis. Carraybound (C array bound) (Gao, et al., 2016) is a static analysis framework to perform array bounds checking based on taint analysis and data-flow analysis to find the indexes that have not been checked against the bounds of the array in C programs.

Le & Soffa (2008) present a framework that automatically generates context-, flow-sensitive analysis to detect user-specified faults. The framework consists of a specification technique that expresses faults and information needed for their detection, a scalable, flow-sensitive algorithm, and a generator that unifies the two. The analysis produced identifies not only faults but also the path segments where the root causes of a fault are located. The same authors (Le & Soffa, 2011) extended their technique to support flow-sensitive symbolic analysis to reduce the number of false positives. Vujosevic-Janicic (2008) presents a static, flow- and context-sensitive system for detecting buffer overflows. The system analyzes the code, generates correctness conditions for commands, and invokes external automated theorem prover (for linear arithmetic) to test the generated conditions.

IntPatch (Zhang, Wang, Wei, Chen, & Zou, 2010) utilizes classic type theory and a data-flow analysis framework to identify potential integer buffer overflow vulnerabilities. It then uses backward slicing to find out related vulnerable arithmetic operations. Finally, it instruments programs with run-time checks. Moreover, IntPatch provides an interface for programmers who want to check integer buffer overflows manually. Bilin, Jiafen, Genqing, Yu, & Dan's (2016) method integrates multiple static analysis tools (ITS4 (Viega, Bloch, Kohno, & McGraw, 2000), Flawfinder (Wheeler, 2018), Splint (Evans & Larochelle, 2002), Cqual (Foster, 2018)) for buffer overflow vulnerabilities detection. The goal is to verify the results, correct mistakes and reduce the false negative and false alarm rate.

Gupta, He, Zhang, & Gupta (2005) and Jeffrey, Gupta, & Gupta, (2008) combine delta debugging and dynamic program slicing to locate faults, which include BO vulnerabilities. First, they use delta debugging to either find a simplified input that induces a failure (failure-inducing input) or isolate a minimal input difference that causes it. Next, they compute the intersection of the statements in the forward dynamic slice of the failure-inducing input and the backward dynamic slice of the faulty output to locate the likely faulty code. Exterminator (Novark, Berger, & Zorn, 2007) is a system that detects, isolates, and corrects two classes of heap-based memory errors (buffer overflow and dangling pointers). It dumps a heap image that contains the complete state of the heap. Buffer overflows and dangling pointer errors can be distinguished because they tend to produce different patterns of heap corruption. Exterminator utilizes a probabilistic algorithm to identify heap corruption associated with buffer overflow and dangling pointers. Both approaches can be combined with crashing techniques to locate the BO vulnerability.

3.2.4. Computational Intelligence

Computational intelligence (CI) encompasses techniques and methods inspired in natural processes that utilize data and experimental observation to derive their actions. CI is suited to problems for which there is inexact and incomplete knowledge and adaptive control actions are required. Examples of CI techniques are *evolutionary computing*, *fuzzy logic*, *genetic algorithms*, *machine learning*, and *neural networks*.

Table 3 maps the CI techniques to the their respective studies. In general, genetic algorithms and evolutionary computing are fitted to support dynamic techniques (e.g., testing) for BO detection. Fuzzy logic and machine learning, in turn, are used to support the static analysis of the code, excepting the study of Padmanabhuni & Tan (2016) in which machine learning is used to create test inputs. The studies whose main technique is based on computational intelligence are described next.

Genetic algorithms and evolutionary computing have been combined with program analysis techniques (e.g., program slicing, taint analysis) to craft tests to detect BO vulnerabilities. Grosso,

Table 3. Computational intelligence techniques used in the studies

Technique	Studies	Qty.
Genetic algorithms & Evolutionary computing	(Grosso et al., 2008; Rawat & Mounier, 2010)	2
Fuzzy logic	(Shahriar & Zulkernine, 2011)	1
Machine learning	(Padmanabhuni & Tan, 2016; Padmanabhuni & Tan, 2015a; Padmanabhuni & Tan, 2014; Meng et al., 2016)	4

Antoniol, Merlo, & Galinier (2008) propose a technique to generate input test data to detect BO vulnerabilities that is fully automated. They combine genetic algorithms, linear programming, and evolutionary testing (Tonella, 2004). The genetic search towards the BO detection relies on a fitness function that takes into account static and dynamic information. Rawat & Mounier (2010) utilize a similar approach. They generate string-based inputs to detect buffer overflow vulnerability in C code. The approach is a combination of genetic algorithm and evolutionary strategies. Program slicing is the static analysis component of the approach. It generates a tainted path from source (malicious inputs) to sink (vulnerable statement). Their fitness function also integrates dynamic and static information.

Shahriar & Zulkernine (2011) utilize fuzzy logic to audit source code. Their fuzzy logic-based code auditing approach is intended to generate improved warnings and to assess a program's overall BO vulnerability level. They defined characteristics related to BO vulnerabilities, developed fuzzy sets for these characteristics, designed fuzzy inference rules to derive warnings, and developed a multi-unit fuzzy logic-based system to assess the quality of programs for BO vulnerabilities.

Padmanabhuni & Tan (2015a) combine static analysis and machine learning for predicting buffer overflow vulnerabilities of x86 executables. Basically, they utilize taint analysis from binary code to identify attributes of BO vulnerabilities. Then they utilize machine learning algorithms to predict where the BO vulnerabilities are in the code. Four well-known machine learning algorithms were assessed: Naive Bayes (NB), Multi-Layer Perceptron (MLP), Simple Logistic (SL) and Sequential Minimum Optimization (SMO). The best classifier was SMO. A similar approach (Padmanabhuni & Tan, 2014) is applied in programs written in C and C++; the recall and precision of the reports are, respectively, 95% and 80.9% for a tree classification algorithm. Meng, Wen, Feng, & Tang (2016) utilize a semi-supervised machine learning algorithm to predict buffer overflow. They analyze every function to extract a 22-attribute vector that is fed to the machine learning algorithm to suggest candidate vulnerable functions.

3.2.5. Symbolic Execution

Symbolic execution is utilized to statically analyze the code of a program. The idea is to utilize symbolic instead of real input values to "execute" a program (King, 1976). Paths traversed during a possible execution of the program are represented as expressions in terms of the symbolic input values, the operators that occur in the commands of the path, and the outcome of conditional commands (e.g., if, while). Among the applications of symbolic execution are testing (Cadar, et al., 2011), debugging (Hentschel, 2016), and vulnerability detection (Chen & Li, 2009; Kim, Lee, Han, & Choe, 2010; Li, Cifuentes, & Keynes, 2010; Padaryan, Kaushan, & Fedotov, 2015). Nevertheless, symbolic execution has limitations like path explosion (Ma, Yit Phang, Foster, & Hicks, 2011) and difficulty to interact with other environments (e.g., libraries, application program interfaces – API) which restrict the applicability of the technique.

Strictly speaking, symbolic execution techniques could be classified into the program analysis or models categories. Because symbolic execution studies bridge these two categories in a particular way, we created a category of its own. These studies are presented in Table 1 and described as follows.

The selected studies, in general, tackle first the limitations of symbolic execution before applying it. Some works do so by modifying the program. Li, Cifuentes, & Keynes (2010) handle loops and complex program structures using flow-sensitive program analysis. Scalability is achieved by using a simple symbolic value representation, filtering out irrelevant dependencies in symbolic value computation and computing symbolic values on demand. Chen & Li (2009) generate a "normalized" version of the program. This version does not contain "pointers." Additionally, the program is expanded to eliminated loops and function calls so that the program contains only if commands. Once the program is expanded, symbolic execution is used to detect buffer overflow and other memory problems. Ibing (2014) uses, in turn, backtracking of symbolic states instead of state cloning, and extends it with a method for merging redundant program paths, based on live variable analysis (Aho, Lam, Sethi, & Ullman, 2007), to tackle the path explosion problem.

Kim, Lee, Han, & Choe (2010) utilize a two-step approach to utilize symbolic execution in the target program. They begin with a cheaper and imprecise analysis based on abstract interpretation. As a result, many false alarms are often eliminated from the first analysis. Then they apply a precise symbolic execution technique to small areas of the code around the alarms where those alarms are checked for false positives.

Padaryan, Kaushan, & Fedotov (2015) present a concolic (concrete and symbolic) technique that combines symbolic execution and testing to detect BO vulnerabilities. They assess the detected bugs based on the symbolic execution of binary code. For a given set of input data that bring the examined program to an abnormal termination, an exploit is constructed for a widespread type of vulnerabilities, including stack buffer overflow. Mouzarani, Sadeghiyan, & Zolfaghari (2015, 2016), in turn, present a concolic execution-based input test data generator for detecting stack- and heap-based buffer overflow in executable code. First, the program is executed with concrete input data. During the execution, the constraints of the execution path are calculated symbolically. Taint analysis is performed and only the constraints that depend on the tainted data are calculated. Besides the path constraints, the vulnerability constraints are calculated for each executed path. The calculated vulnerability constraints are combined with the path constraints and are queried from a constraint solver. If the solver finds a solution, it will be used to generate data that execute the same execution path and cause an overflow in it. Analogously, Li & Shieh (2011) also propose a concolic technique, called loop-aware concolic execution, for testing software and analyzing loop-related variables. The goal is to improve the performance in the presence of loops. They analyze binary executables in x86 platforms.

Muntean, Kommanapalli, Ibing, & Eckert (2015) approach starts from already detected BO vulnerabilities to generate bug fixes. They generate bug fixes for buffer overflow automatically using symbolic execution, code patch patterns, quick fix locations, user input saturation and Satisfiability Modulo Theories (SMT). Thus, it does not properly detect BO bugs, but it can be used to check possible false positives. BovInspector (Gao, Wang, & Li, 2016) uses symbolic execution to automatically identify those buffer overflow warnings reported by static analysis that are true warnings or false warnings. To avoid path explosion, BovInspector only focuses on the execution paths that cover the buffer overflow warnings. BovInspector automatically repairs these buffer overflow vulnerabilities according to human repair strategies.

3.2.6. Testing

Testing allows BO vulnerabilities to be observed. Successful testing causes the system to crash as a result. When a segmentation violation message is obtained, chances are the program is vulnerable. We identified three testing approaches to detect BO vulnerabilities. One is to craft special inputs to check the buffers' limits. In general, they utilize lightweight program analysis to obtain the limits of the program's buffers to derive test inputs. The second approach is based on mutation testing (DeMillo, Lipton, & Sayward, 1978); it consists in seeding possible BO violations to be checked by test cases. Finally, there are techniques that utilize genetic algorithms, symbolic execution, and

Table 4. Testing techniques used in the studies

Technique	Studies	Qty.
Buffer limits checking	(Larson & Austin, 2003; Bruschi et al., 1998; Chen et al., 2013; Haugh & Bishop, 2003; Padmanabhuni & Tan, 2016; Padmanabhuni & Tan, 2015b)	6
Mutation testing	(Shahriar & Zulkernine, 2008; Woodraska et al., 2011; Ghosh et al., 1998)	3
Test input generation	(Xu et al., 2008; Duraes & Madeira, 2005; Grosso et al., 2008; Rawat & Mounier, 2010; Mouzarani et al., 2015; Mouzarani et al., 2015)	7

machine learning to generate test inputs. All three approaches lead to the generation of test data to detect BO vulnerabilities, varying in how the test data is derived.

Table 4 describes the studies associated with each approach. Other studies that use testing as a subsidiary technique are described with their main techniques, for instance, genetic algorithms (Grosso, Antoniol, Merlo, & Galinier, 2008; Rawat & Mounier, 2010), theorem provers (Mouzarani, Sadeghiyan, & Zolfaghari, 2015), and symbolic execution (Mouzarani, Sadeghiyan, & Zolfaghari, 2016). We detail the studies for which testing is the main technique next.

Larson & Austin (2003) utilize shadow variables for every input to control their limits and, subsequently, to conduct flow-sensitive analysis to create tests that explore the limits of the program data structures. Bruschi, Rosti, & Banfi (1998) use large input parameters to cause a segmentation fault in the program under testing. Then a test program is used to derive the exact layout of a function stack frame and the most critical parameters needed to exploit the code. ARMORY (Chen, et al., 2013) allows the tester to carry out functional testing as usual. The tester executes a process to notify ARMORY to make BO tests for a specific process. For the parent process (original test), the return value is the length of the original input string. On the other hand, for the child process, created by ARMORY by changing Linux system calls, the return value is the length of the corresponding crafted BO test strings. These specially crafted inputs are used to detect BO vulnerabilities.

Duraes & Madeira (2005) propose a strategy that mixes executable code scanning and testing to detect BO vulnerabilities. The goal of the scanning is to locate the code signatures that are related to buffer use and buffer-limit check omission. The functions containing the signatures are deemed as "suspect." All functions tagged as "suspect" are tested by supplying values to the function parameters that are likely to cause a buffer overflow. The values supplied to the parameters are specifically intended to cause buffer overruns and are based on the knowledge obtained during the code search.

STOBO (Systematic Testing of Buffer Overflow) (Haugh & Bishop, 2003) is a tool to support BO detection. The tester writes a specification associated with code locations in a specific-domain language. The program being tested is then instrumented according to the specification. During testing, possible violations of the memory allocated are checked during the execution of test cases. Padmanabhuni & Tan (2015b) use static analysis to identify constraints on inputs and use a set of rules based on buffer size to generate test cases. They use the control and data dependency information of the sink to identify inputs affecting the sink and predicates performing input checks. The same authors (Padmanabhuni & Tan, 2016) use machine learning techniques to predict whether automated test inputs should be generated for potentially vulnerable statements. They collect static code attributes capturing input validation and data size checking mechanisms to feed the classifiers. The goal of such test inputs is to trigger buffer overflows and confirm their vulnerability.

Mutation testing was the underlying technique used in several approaches (Ghosh, O'Connor, & McGraw, 1998; Shahriar & Zulkernine, 2008; Woodraska, Sanford, & Xu, 2011). They emulate the effect of flaws in software by using data perturbation functions, better known as fault injection functions or mutation operators. If the simulated flaw violates the security of the system, then the location where the flaw was introduced is identified for further investigation.

Table 5. Models used in the studies

Models	Studies	Qty.
Annotations	(Larochelle & Evans, 2001; Hackett et al., 2006; Evans & Larochelle, 2002)	3
Assertions	(Dor et al., 2003; Dor et al., 2001; Haugh & Bishop, 2003)	3
Attack trees	(Woodraska et al., 2011)	1
Logical expressions	(Hart et al., 2008; Weber et al., 2001; Chess, 2002; Ferrara et al., 2008; Chen et al., 2004; Ganapathy et al., 2003; Vujosevic-Janicic, 2008)	7
Finite state machines	(Liang, 2009; Chen et al., 2003)	2
Petri Nets	(Woodraska et al., 2011)	1

Other approaches (Xu, Godefroid, & Majumdar, 2008) rely on test input generation techniques to address BO detection. Xu, Godefroid, & Majumdar (2008) propose a test input generation algorithm that tracks and symbolically reasons about lengths of input buffers and strings. This is done by extending symbolic execution with respect to buffer contents to also include its length. To speed up the search and make it more tractable, symbolic execution only tracks the influence of data values stored in prefixes of input buffers, instead of full buffers.

3.2.7. Models

The approaches in this category define a model from which BO detection follows through. We adopt a broad concept of model: it can be annotations or assertions in the program; finite state machines, petri nets, or attack trees derived from it; or logical expressions that allow verifications or proofs to be carried out. Therefore, the approaches are based on some form of model from which actions are performed to identify BO vulnerabilities.

Table 5 lists the models used in the studies. Models were used in 17 different studies. Six different models were identified: annotations, assertions, attack trees, logical expressions, finite state machines, and petri nets. The differences among annotations, assertions and logical expressions are subtle. We classified the model as annotations when the authors mentioned this way or when they were included as comments in the code. Assertions were associated with models checked at run-time and logical expressions when they were statically verified by either a model checker or a theorem prover. The studies that use models as main strategy to locate BO vulnerabilities are described next.

Larochelle & Evans (2001) exploit semantic comments (i.e., annotations) added to source code to enable local checking of inter-procedural properties. They then utilize lightweight static checking techniques to obtain good performance and scalability, though, sacrificing soundness and completeness. Hart, Ku, Gurfinkel, Chechik, & Lie (2008) define proof templates designed to work when a program uses common structures to traverse an array. A model checker is used to prove the original property using these templates extended by predicates and assumptions. Hackett, Das, Wang, & Yang (2006) purpose a modular checker to statically find and prevent every buffer overflow. Lightweight annotations specify requirements for safely using each buffer, and functions are checked individually by a model checker to ensure they obey these requirements and do not overflow. Chen, Dean, & Wagner (2004) describe an empirical study in which MOPS, a tool for software model checking security-critical applications, is used to verify six large, well-known, frequently used, open-source packages.

Liang (2009) detects vulnerabilities using an extended finite state machine. The security state of a variable is identified by a property set that may consist of multiple security-related properties. A fine-grained state transition is provided to support accurate recognition of program security-related behaviors. Taint analysis is used to avoid neglecting tainted data sources and to prevent false negatives results. Chen, Kalbarczyk, Xu, & Iyer (2003) introduce a finite state machine (FSM)

modeling methodology capable of expressing the exploitation of an object. They model an operation on an object as a series of primitive FSMs. The FSM model allows to reason whether a vulnerability is not present in the implementation. During this process, one can uncover a previously unknown vulnerability. Woodraska, Sanford, & Xu (2011) create program mutants for assessing security tests from threat models represented by attack trees and Petri nets.

Mjolnir (Weber, Shah, & Ren, 2001) is a buffer overflow analysis tool for generating and solving safety constraints. For every call to a string library function, such as strcpy() or strcat(), a constraint is generated which summarizes the effect of that function call on its arguments. Then all of these constraints are grouped and solved with a fixed-point algorithm. Mjolnir builds and solves constraint sets in an attempt to prove that each potential vulnerability is not overflowable. If it fails to do so, the potential vulnerability must be examined by other means, such as testing, to determine whether it is indeed overflowable. Eau Claire (Chess, 2002) translates a program's source code into a series of verification conditions and presents the verification conditions to an automatic theorem prover. If it refutes the theorem, then the associated function is in violation of one or more of the specifications. A counterexample provided contains enough information so that the user can track down the source of the mismatch.

Ganapathy, Jha, Chandler, Melski, & Vitek (2003) combine program slicing, points-to and taint analysis to generate linear constraints associated with buffer declarations, assignments, function calls, and return statements. Then they use linear programming to obtain values for the constraint variables. The goal is to obtain the best possible estimate of the number of bytes used and allocated for each buffer in any execution of the program. Based on the values inferred by the solver, the detector decides whether there was an overrun on each buffer. Dor, Rodeh, & Sagiv (2003; 2001) utilize integer analysis to identify potential violations of the code. The programmer should annotate every procedure with a precondition, a post condition, and its potential side effects. The annotated program goes through a source-to-source semantic-preserving transformation. The generated program yields a run-time error whenever a contract is violated. When a run-time error occurs, pointer interactions are analyzed to collect point-to information. Then the procedure's code and point-to information are used to generate a procedure that manipulates integers. The resultant integer program is analyzed using a conservative integer-analysis algorithm to determine all potential violations.

Ferrara, Logozzo, & Fähndrich (2008) statically analyze programs compiled to the MSIL (Microsoft Intermediate Language) instruction set to check memory safety. It statically checks contracts and use them to refine the precision of the analysis, e.g. by exploiting preconditions. To achieve precision and scalability, the authors utilize abstract interpretation based on abstract domains (memory regions) to check the safety of read and write operations.

3.2.8. Code Inspection

Code inspection in this review comprehends techniques that find vulnerabilities by reading the source code. The approaches in this category aims at improving the process of reading code. Broadly speaking, any technique reporting vulnerabilities would fall in this category. However, we preferred to select only techniques whose propose was to support source code reading, which narrowed down to only three studies as described in Table 1.

Chen & Mao (2012) propose a game to support code inspection to locate BO vulnerability. Wu, Siy, & Gandhi (2011) define generalized patterns of relationship between software elements, and their association with known vulnerabilities. The patterns are derived from the Common Weakness Enumeration (CWE), a community-driven taxonomy of software weaknesses (MITRE, 2017). These patterns can be used for inspection and verification of the code.

DaCosta, Dahn, Mancoridis, & Prevelakis, (2003) hypothesize that a small percentage of functions near a source of input (e.g., file I/O) are the most likely to contain a security vulnerability. They refer to these functions as FLFs (Front Line Functions), and the percentage of functions likely to contain a security vulnerability as the FLF density. They developed a tool, called FLF Finder tool, to identify

areas of high vulnerability likelihood automatically. FLF tool was developed to aid code inspectors to focus on the more vulnerable excerpts of code.

4. DISCUSSION

The discussion below is guided by the research question established in the *Systematic Literature Review* Section.

4.1. R1. Which are the Relevant Vulnerabilities Related to Buffer Overflow?

During the analysis of the selected works, we identified three basic BO vulnerabilities, namely, stack-based, heap-based, and integer buffer overflow. The majority of the studies either targets all BO types without distinction or focuses on memory errors in general, being BO just another one. Few of them target specific BO types: stack-based overflow is mentioned in (Ghosh, O'Connor, & McGraw, 1998; Mouzarani, Sadeghiyan, & Zolfaghari, 2016; Padaryan, Kaushan, & Fedotov, 2015), integer buffer overflow is the main or part of the goal of the techniques described in (Chen, Kalbarczyk, Xu, & Iyer, 2003; Yamaguchi, Golde, Arp, & Rieck, 2014; Zhang, Wang, Wei, Chen, & Zou, 2010; 2011) and heap-based buffer overflow is addressed in (Mouzarani, Sadeghiyan, & Zolfaghari, 2015; Novark, Berger, & Zorn, 2007).

Thus, only 9 studies out of 67 are focused on particular BO exploits. Many studies target different memory errors (e.g., memory leaks, null pointer dereferention, unsafe memory access) (Dor, Rodeh, & Sagiv, 2003; Ferrara, Logozzo, & Fähndrich, 2008; Larochelle & Evans, 2001; Le & Soffa, 2011) or several vulnerabilities (Akbari, Berenji, & Azmi, 2010; Le & Soffa, 2011; Li & Shieh, 2011; Liang, 2009; Livshits & Lam, 2003; Pozza & Sisto, 2008; Viega, Bloch, Kohno, & McGraw, 2000; Woodraska, Sanford, & Xu, 2011) (Wu, Siy, & Gandhi, 2011). In both cases, BO is included, but it is not the main goal.

The drawback of very general techniques is that they lead the practitioner to search a list of issues that may not be related to BO. Thus, there is a lack of techniques specifically designed for BO detection. Fulfilling this gap requires techniques that take into account the hardware running the system, especially to tackle stack-based and integer buffer overflow.

4.2. R2. How Have the Buffer Overflow Detection Techniques Evolved?

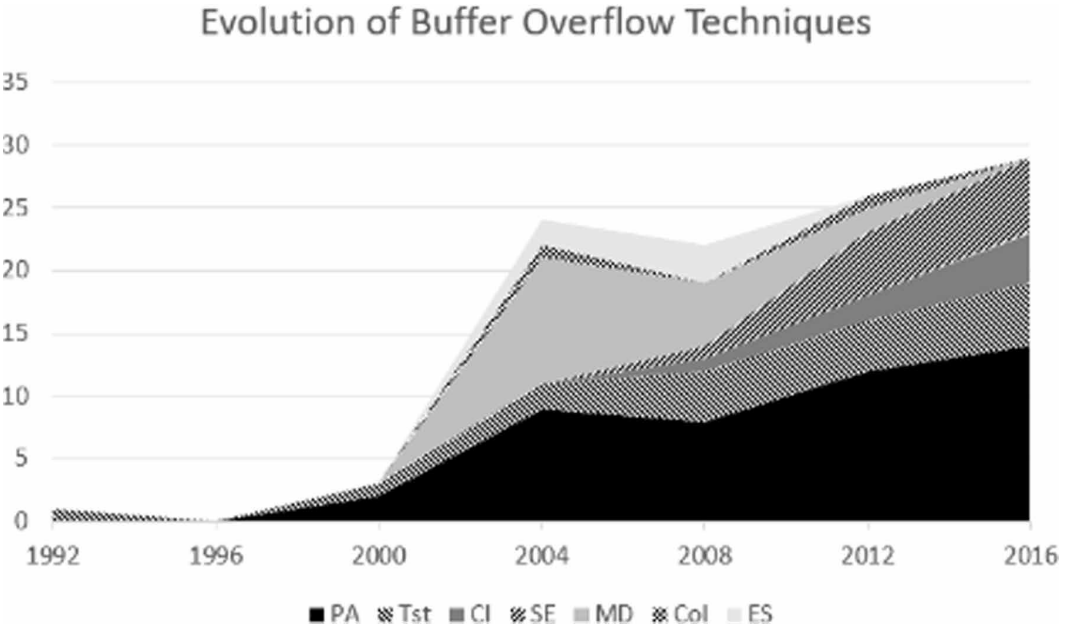
BO detection techniques have prevalently aimed at reporting vulnerable locations. Reporting is the main or subsidiary objective of 61 studies. Crashing is the only target of six studies; other five studies have crashing and reporting purposes (see Figure 5). Repairing vulnerable locations is the goal of recent and restricted studies (Gao, Wang, & Li, 2016; Muntean, Kommanapalli, Ibing, & Eckert, 2015; Novark, Berger, & Zorn, 2007; Zhang, Wang, Wei, Chen, & Zou, 2010). The ideal technique crashes the system to confirm the existence of a vulnerability; then it locates and repairs the vulnerability. In doing so, it achieves the ultimate goal of producing BO-free software.

Source code analysis is utilized in most studies (60 out 67). However, object code analysis generates hardware-specific techniques which can lead to techniques targeted to BO detection, as suggested in the discussion of RQ1. Indeed, new studies (Padmanabhuni & Tan, 2015a; 2015b) combine object code analysis with computational intelligence, program analysis, and testing techniques.

Figure 7 describes the evolution of techniques to tackle BO detection. It aggregates studies in a five-year time span. For example, at the 2004 mark, representing the time span from 2001 to 2004, there were 24 studies in total, being nine based on program analysis, two on testing, ten on models, one on code inspection, and two empirical studies.

Studies targeting BO detection began to gain momentum from the years 2000s on. Program analysis constitutes the main technique for BO detection. The number of studies during the years either increases or flattens, but never decreases. The reason is because program analysis is an enabler for the application of other techniques when it is not the main technique as shown in Figure 6. Testing has a

Figure 7. Evolution of the techniques to detect BO vulnerabilities



similar behavior since the number of papers keeps growing, not as expressively as program analysis, during the years. Symbolic execution and computational intelligence have presented a growing interest by researchers from 2012 due mainly to recent improvements in the techniques. On the other hand, models and code inspection show a diminished interest in the last years. A possible explanation is because they require special conditions to scale well to real applications.

Figure 7 shows that program analysis, testing, computational intelligence and symbolic execution are on the raise. Likely because the new studies utilize these techniques in combination, which indicates a promising approach to tackle BO vulnerabilities.

4.3. R3. Which are the Techniques Available to Detect Buffer Overflow Vulnerabilities Before the Release of The Software?

The techniques for BO detection were described and organized in categories in Section *Results*. Practitioners, though, are interested in techniques that are available for use or that are promising. We address these questions by discussing the strong and weak points of the categories of techniques. They are summarized in Table 6. Three categories, namely, program analysis, testing, and code inspection, have techniques available for use by practitioners. The other categories include promising techniques that can leverage program analysis testing, and code inspection techniques.

Because buffer overflow is a code weakness, techniques that handle code, i.e., program analysis techniques, were the primary focus of research. Lightweight approaches, based on compiling techniques, that search for BO-related patterns in code, have evolved to handle real-world software (Yamaguchi, Golde, Arp, & Rieck, 2014). However, they can report too many issues that overwhelm the practitioners. Techniques that carry out a thorough static analysis (e.g., data-flow analysis, abstract interpretation) lead to less false alarms. Points-to and taint analysis are specializations of static analysis that detect more vulnerabilities than compiling techniques with less false positives (Gao, et al., 2016; Livshits & Lam, 2003). Despite of being costly, these techniques are able to scan large programs; however, the number of false positives can still be not manageable by practitioners.

Dynamic techniques, for instance, dynamic slicing and delta debugging, are able to determine the root cause of BO vulnerabilities (Gupta, He, Zhang, & Gupta, 2005). Nevertheless, they depend on program runs to collect data. If too many runs are needed, the techniques can become costly (Gupta, He, Zhang, & Gupta, 2005; Jeffrey, Gupta, & Gupta, 2008). One possible solution is to apply dynamic techniques during testing.

Program analysis techniques can be utilized by the practitioner provided he or she deals with false positives. This limitation, however, hinders the application of program analysis; the study of Fang & Hafiz (2014) indicates that the use static analysis for BO detection is uncommon in practice. BO specific techniques and techniques that better rank the vulnerable locations can make program analysis techniques practical.

Testing, especially penetration testing, is used to detect BO vulnerabilities in practice (Fang & Hafiz, 2014). A test should overwrite control sensitive data structures to crash a program (Chen, et al., 2013). Thus, testing does not locate BO vulnerabilities; it detects their presence. To devise such a test, the tester needs to be knowledgeable of the hardware of the system (Bruschi, Rosti, & Banfi, 1998). Currently, testing is conducted in *ad hoc* manner; practitioners rely on fuzz testing to detect BO vulnerabilities (Fang & Hafiz, 2014). To improve the effectiveness of testing in practice, it should be combined with other techniques to systematically generate tests to detect BO vulnerabilities.

There are approaches that combine symbolic execution and computational intelligence to develop effective tests and to locate BO vulnerabilities (Padmanabhuni & Tan, 2016; Rawat & Mounier, 2010; Shahriar & Zulkernine, 2011; Xu, Godefroid, & Majumdar, 2008). Additionally, fault localization (Gupta, He, Zhang, & Gupta, 2005) and program repair (Gao, Wang, & Li, 2016) can utilize testing results to obtain a complete BO detection: from the system crash by means of tests, the technique would also locate and fix the vulnerable location.

Code inspection requires the practitioner to inspect the code to locate vulnerabilities. Reading techniques can improve the practitioner's performance, especially those that indicate more vulnerable code (DaCosta, Dahn, Mancoridis, & Prevelakis, 2003; Wu, Siy, & Gandhi, 2011). However, they dependent on the practitioners' skill and on the team's adherence to a development process that includes them. Code inspection will hardly be adopted if the practitioners were demanded to scan a large software already coded.

Symbolic execution benefits from the combination of techniques like program analysis and testing. These techniques are utilized to weed out irrelevant paths, data or states so that the symbolic execution only tracks information that leads to identify a BO vulnerability. Simplified symbolic execution can analyze large programs with a false positive rate of 10% (Li, Cifuentes, & Keynes, 2010). Testing is utilized as a precursor to identify the existence of a BO vulnerability (Padaryan, Kaushan, & Fedotov, 2015) or to reduce the number of constraints to be calculated (Mouzarani, Sadeghiyan, & Zolfaghari, 2015). It is utilized to confirm the existence of a BO vulnerability detected (Padaryan, Kaushan, & Fedotov, 2015).

Computational intelligence leverages other techniques like program analysis and testing. When it is combined with these techniques the results are improved (Padmanabhuni & Tan, 2014; 2015a; Rawat & Mounier, 2010; Shahriar & Zulkernine, 2011). Genetic algorithms and evolutionary testing create input test data that comprise the restrictions established by the code and the hardware of the system. Fitness functions that guide the search utilize static and dynamic information collected using program analysis (Rawat & Mounier, 2010).

Machine learning, in turn, can predict the more vulnerable locations. In doing so, they improve the performance of program analysis techniques. However, some issues need to be addressed: Which is the best machine learning algorithm for BO detection? How to train it for practical application?

Techniques based on models require that assertions, annotations or logic expressions be developed. They can be created automatically or by the practitioner. Then these models are checked at run-time or verified statically by a model checker or theorem prover to detect BO vulnerabilities. Most model-based studies present lower false positive rates, but they do not show feasibility at industrial settings. Studies

Table 6. Strong and weak points of the BO detection techniques

Technique	Discussion
Program Analysis	**Strong points:** Compiling techniques scale well to detected suspicious patterns; program transformation is used to fix vulnerabilities; static analysis techniques generate less false alarms than compiling techniques (Wagner, Foster, Brewer, & Aiken, 2000; Pozza & Sisto, 2008); points-to and taint analysis are specializations of static analysis (Livshits & Lam, 2003; Gao, et al., 2016); delta debugging and dynamic program slicing determine the root cause of BOs (Gupta, He, Zhang, & Gupta, 2005). **Weak points:** Program analysis techniques still produce a high number of false alarms; delta debugging and dynamic slicing can be costly (Gupta, He, Zhang, & Gupta, 2005; Jeffrey, Gupta, & Gupta, 2008).
Testing	**Strong points:** Ease adoption by practitioners; used in real-world programs; can be combined with computational intelligence techniques and symbolic execution. **Weak points:** Do not locate vulnerabilities; running tests can be costly; requires knowledge of the hardware architecture.
Symbolic Execution	**Strong points:** Can analyze large programs with a false positive rate of 10% (Li, Cifuentes, & Keynes, 2010); benefits from the combination of techniques like program analysis and testing to weed out irrelevant paths, data or states (Padmanabhuni & Tan, 2016; Kim, Lee, Han, & Choe, 2010; Padaryan, Kaushan, & Fedotov, 2015). **Weak points:** Particular programs can achieve 40% of false positives; needs more experiments with real programs.
Computational Intelligence	**Strong points:** Leverages other techniques like program analysis and testing (Rawat & Mounier, 2010; Shahriar & Zulkernine, 2011; Shaw, Doggett, & Hafiz, 2014; Padmanabhuni & Tan, 2016; Padmanabhuni & Tan, 2014). **Weak points:** Techniques assessed by small programs (Padmanabhuni & Tan, 2015a).
Code Inspection	**Strong points:** Can be added to the software process; static tools and patterns can indicate vulnerable code to be inspected (Wu, Siy, & Gandhi, 2011). **Weak points:** Hard to scale for large software; it depends on the practitioners' skill; does not automatically locate vulnerabilities.
Models	**Strong points** Detects vulnerabilities with lower false positives; promising studies tackle large software; it can be used in integrated development environments (Chen, Dean, & Wagner, 2004; Ferrara, Logozzo, & Fähndrich, 2008). **Weak points** Needs contracts written by programmers to scale (Ferrara, Logozzo, & Fähndrich, 2008).

that tackle large software suggest models should be written by programmers during development (Ferrara, Logozzo, & Fähndrich, 2008). Thus, they require extra tasks during development; it remains to be shown whether programmers would embrace these new tasks.

The results suggest that the techniques available for BO detection are not ready for adoption at industrial settings. Much progress was made to tackle production-size software and to reduce the number of false positives (see discussion on R4); however, their adoption depends on how the techniques improve the practitioners' tasks.

New studies combine approaches to make the results useful for practitioners. Program analysis can be improved by BO-specific techniques (see the discussion on R1) and by techniques (e.g., computational intelligence, models) that highlight the most vulnerable locations. Symbolic execution

and computational intelligence techniques can leverage testing by creating effective input data for BO detection. As a result, they provide guidance to a current ad hoc task. However, how these new techniques perform in practice remains to be shown.

4.4. R4. Which are the Techniques Scalable to Be Utilized at Industrial Settings?

In this section, we discuss techniques scalable for use at industrial settings. We aimed at identifying techniques that can handle programs with 10 kLOC or more with a low false alarm rate. The idea is that 10kLOC are a manageable chunk of code for a programmer. If the technique can analyze it in reasonable amount of time (up to 10 minutes) with a fairly reduced number of false alarms (up to 10%) they can be added in his or her programming practice.

Program analysis techniques can handle programs of 10 kLOC or more; indeed, static analysis techniques can carry out precise analysis of large programs (up to 600kLOCs) with less false alarms than compiling techniques (Pozza & Sisto, 2008; Wagner, Foster, Brewer, & Aiken, 2000). However, they do not meet the requirement of a 10% false alarm rate.

On the other hand, testing is not limited by the size of the system under assessment. However, it is in general applied in *ad hoc* manner, which can become costly. When testing is combined with other techniques (e.g., program analysis, symbolic execution, and computational intelligence) it is limited by the restrictions of these techniques. Code inspection is scalable at industrial settings if the target is 10 kLOC programs and reading techniques are part of the software process. Nevertheless, the costs of reading techniques should be assessed in practice because skillful practitioners are an expensive asset.

Simplified symbolic execution can analyze large programs with a false positive rate of 10%; however, it achieves 40% of false positives for particular programs (Li, Cifuentes, & Keynes, 2010). Techniques combining static analysis to eliminate false alarms and precise symbolic execution on most suspicious code excerpts tackle programs around 10kLOC (Kim, Lee, Han, & Choe, 2010). Concolic techniques (testing combined with symbolic execution) presented interesting results for real programs (Padaryan, Kaushan, & Fedotov, 2015). Thus, simplified symbolic execution combined with testing is on track to handle 10 kLOC programs, but the 10% false alarm rate still needs to be verified in practice.

Computational intelligence is useful to improve program analysis and testing. However, they were verified in small programs (Padmanabhuni & Tan, 2015a). Likewise, most model-based techniques were assessed in small programs and the tools require an amount of time that would not allow its adoption in software practice (Chen, Kalbarczyk, Xu, & Iyer, 2003; Chess, 2002; Dor, Rodeh, & Sagiv, 2003; Hart, Ku, Gurfinkel, Chechik, & Lie, 2008; Larochelle & Evans, 2001; Liang, 2009; Weber, Shah, & Ren, 2001; Woodraska, Sanford, & Xu, 2011). The techniques that tackle large software (Chen, Dean, & Wagner, 2004; Ferrara, Logozzo, & Fähndrich, 2008) should be verified whether they reach a small false alarm rate (less than 10%) at a feasible time-frame.

Thus, the current techniques fail to meet the scalability framework established (10 kLOC program-size, 10% false alarm rate, 10 minutes execution time), which explain in part why they are rarely used in practice (Fang & Hafiz, 2014). Nevertheless, this framework serves as a target to be aimed by new techniques. Those that meet its requirements tend to be practical.

4.5. Recommendations

From the results of our SLR, we devised recommendations to practitioners and researchers alike. Our suggestions to practitioners are presented next.

1. **Utilize a program analysis technique.** There are techniques (e.g., static analysis, taint analysis) that thoroughly analyze 10 kLOC programs in affordable time-frame (less 10 minutes). By using them, the practitioners will have to manage less false alarms.

2. **Generate tests to check the more suspicious issues.** With the results of program analysis, the practitioners will be able to develop more systematic penetration tests using his or her knowledge of the system's hardware.
3. **Add code review techniques to the software development process.** When adding the code reading task, make sure the practitioners are knowledgeable of the main BO code vulnerabilities patterns of the Common Weakness Enumeration (CWE) (MITRE, 2017); and make use of techniques to focus the practitioner's attention to the most vulnerable locations.

The suggestions to researchers are described as follows.

1. **Design BO-specific techniques.** These techniques focus the practitioner's attention to BO issues. To achieve this goal, they should be aware of the system's hardware.
2. **Develop complete BO detection techniques.** The ultimate goal of BO detection is to generate BO-free software. That requires crashing the system, locating its cause, and fixing it. The techniques in this survey address one or at most two of these aims. Techniques addressing all goals will entice the practitioners to use them.
3. **Combine different categories of techniques.** Computational intelligence and symbolic execution have shown promising results in identifying more vulnerable locations and more effective input tests. However, more effort should be directed to identify the best algorithms for BO detection so that the techniques can tackle production-size software.
4. **Aim at scalable techniques.** Techniques that tackle 10 kLOC programs, with 10% false alarm rate, and produce results in 10 minutes are prone to be adopted in the software development process.
5. **Conduct empirical studies at development settings.** Promising techniques should be assessed at development sites to understand how they can be better used by practitioners.

5. CONCLUSION

Despite being one of the first exploited code vulnerabilities, buffer overflow (BO) still threatens the security of systems that were written or utilize components written in vulnerable programming languages such as C and C++. We carried out a comprehensive systematic review of the literature (SLR) to identify which are the techniques available to the practitioner for BO detection during software development.

The SLR reviewed 67 studies, including relevant works referred to in the primary studies (snowballing). It was guided by four research questions: (1) Which are the relevant vulnerabilities related to buffer overflow? (2) How has buffer overflow detection evolved? (3) Which are the techniques available to detect buffer overflow vulnerabilities before the release of the software? (4) Which are the techniques scalable to be utilized at industrial settings?

We found three main BO vulnerabilities, namely, stack-based, heap-based and integer buffer overflow. Most of the studies address many vulnerabilities (e.g., format string, BO) or memory errors, being not specific to BO detection. To assess the techniques proposed in the studies, we organized them in six categories: program analysis, testing, computational intelligence, symbolic execution, models, and code inspection. Program analysis techniques, testing and code inspection are techniques available for use by the practitioner. However, program analysis adoption is hindered by the high number of false alarms; testing is broadly used but in *ad hoc* manner; and code inspection can be used in practice provided it is added as a task of the software development process.

We defined a scalability framework to assess the techniques' practicality. They should tackle 10 kLOC programs, produce at most 10% false alarm rate, and generate the result in up to 10 minutes. The current techniques fail to meet the scalability framework, which explain in part why they are

rarely used in practice (Fang & Hafiz, 2014). New techniques combining object analysis, program analysis and testing with symbolic execution and computational intelligence seem promising for practical BO detection.

From the SLR's results, recommendations to tackle BO vulnerabilities in practice and to direct the research in BO detection were devised. As a concluding remark, we hope the results presented in this paper help practitioners and researchers to adopt and develop efficient techniques to detect BO vulnerabilities at industrial settings.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

Akbari, M., Berenji, S., & Azmi, R. (2010). Vulnerability detector using parse tree annotation. In *International Conference on Education Technology and Computer (ICETC)* (pp. 257-261). doi:10.1109/ICETC.2010.5529688

Bilin, S., Jiafen, Y., Genqing, B., Yu, Z., & Dan, S. (2016). A static comprehensive analytical method for buffer overflow vulnerability detection. In *International Conference on Computer Science and Electronic Technology (CSET) (*pp. 151-155).

Binkley, D. (2007). *Source Code Analysis: A Road Map. In Future of Software Engineering (FOSE)* (pp. 104–119). IEEE Computer Society.

Bruschi, D., Rosti, E., & Banfi, R. (1998). A Tool for Pro-active Defense Against the Buffer Overrun Attack. In *European Symposium on Research in Computer Security (ESORICS)* (pp. 17-31). Springer. doi:10.1007/BFb0055853

Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., & Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 1066-1071). doi:10.1145/1985793.1985995

Chen, H., Dean, D., & Wagner, D. A. (2004). Model Checking One Million Lines of C Code. In *Network and Distributed System Security Symposium (NDSS).* The Internet Society.

Chen, J., & Mao, X. (2012). Bodhi: Detecting Buffer Overflows with a Game. In *2012 IEEE Sixth International Conference on Software Security and Reliability Companion (SERE-C)* (pp. 168-173).

Chen, L.-H., Hsu, F.-H., Hwang, Y., Su, M.-C., Ku, W.-S., & Chang, C.-H. (2013). ARMORY: An automatic security testing tool for buffer overflow defect detection. *Computers & Electrical Engineering*, *39*(7), 2233–2242. doi:10.1016/j.compeleceng.2012.07.005

Chen, S., Kalbarczyk, Z., Xu, J., & Iyer, R. K. (2003). *A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities. In Dependable Systems and Networks (DSN)* (pp. 605–614). IEEE Computer Society.

Chen, S., & Li, Z. (2009). Simplifying Buffer Overflow Detection Using Site-Safe Expressions. In *ACIS International Conference on Computer and Information Science (ICIS)* (pp. 977-982). IEEE Computer Society. doi:10.1109/ICIS.2009.158

Chess, B. (2002). Improving Computer Security Using Extended Static Checking. In *IEEE Symposium on Security and Privacy* (pp. 160-173). IEEE Computer Society. doi:10.1109/SECPRI.2002.1004369

Constantin, L. (2012). Artema Hybrid Point-of-sale Devices Can Be Hacked Remotely, Researchers Say. *PCWorld*.

DaCosta, D., Dahn, C., Mancoridis, S., & Prevelakis, V. (4 de 6 de 2003). Characterizing the 'Security Vulnerability Likelihood' of Software Functions. In *International Conference on Software Maintenance (ICSM)* (p. 266). IEEE Computer Society.

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, *11*(4), 34–41. doi:10.1109/C-M.1978.218136

Ding, B., He, Y., Wu, Y., Miller, A., & Criswell, J. (2012). *Baggy Bounds with Accurate Checking* (pp. 195–200). ISSRE Workshops.

Dor, N., Rodeh, M., & Sagiv, M. (2001). Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *Static Analysis Symposium* (pp. 194-212). Springer Verlag. doi:10.1007/3-540-47764-0_12

Dor, N., Rodeh, M., & Sagiv, M. (2003). *CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows. In Proc. Programming Language Design and Implementation (PLDI'03)*. ACM Press. doi:10.1145/781131.781149

Drayton, P., Albahari, B., & Merrill, B. (2001). *C# Essentials.* Sebastobol, CA: O'Reilley & Associates.

Duraes, J., & Madeira, H. (2005). A Methodology for the Automated Identification of Buffer Overflow Vulnerabilities in Executable Software Without Source-Code. In *Latin-American Symposium on Dependable Computing (LADC)* (pp. 20-34). Springer. doi:10.1007/11572329_5

Erlingsson, Ú. (2007). Low-Level Software Security: Attacks and Defenses. In A. Aldini & R. Gorrieri (Eds.), *Foundations of Security Analysis and Design IV: FOSAD 2006/2007 Tutorial Lectures* (pp. 92–134). Springer Berlin Heidelberg; doi:10.1007/978-3-540-74810-6_4

Evans, D., & Larochelle, D. (2002). Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, *19*(1), 42–51. doi:10.1109/52.976940

Fang, M., & Hafiz, M. (2014). *Discovering buffer overflow vulnerabilities in the wild: an empirical study. In Empirical Software Engineering and Measurement (ESEM)* (pp. 1–10). ACM. doi:10.1145/2652524.2652533

Ferrara, P., Logozzo, F., & Fähndrich, M. (2008). Safer unsafe code for. NET. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (pp. 329-346). ACM.

FosterJ. (2018). Cqual. Retrieved from http://www.cs.umd.edu/~jfoster/cqual/

Ganapathy, V., Jha, S., Chandler, D., Melski, D., & Vitek, D. (2003). Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proc. ACM Conference on Computer Security (CCS'03).* ACM. doi:10.1145/948109.948155

Gao, F., Chen, T., Wang, Y., Situ, L., Wang, L., & Li, X. (2016). *Carraybound: static array bounds checking in C programs based on taint analysis. Internetware* (pp. 81–90). ACM. doi:10.1145/2993717.2993724

Gao, F., Wang, L., & Li, X. (2016). BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 786–791).

Ghosh, A. K., O'Connor, T., & McGraw, G. (1998). An Automated Approach for Identifying Potential Vulnerabilities in Software. In *IEEE Symposium on Security and Privacy* (pp. 104-114). IEEE Computer Society. doi:10.1109/SECPRI.1998.674827

Grosso, C. D., Antoniol, G., Merlo, E., & Galinier, P. (2008). Detecting buffer overflow via automatic test input data generation. *Computers & Operations Research*, *35*(10), 3125–3143. doi:10.1016/j.cor.2007.01.013

Gupta, N., He, H., Zhang, X., & Gupta, R. (2005). Locating faulty code using failure-inducing chops. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (pp. 263–272).

Hackett, B., Das, M., Wang, D., & Yang, Z. (2006). Modular checking for buffer overflows in the large. In *International Conference on Software Engineering (ICSE)* (pp. 232-241). ACM. doi:10.1145/1134285.1134319

Harman, M. (2010). *Why Source Code Analysis and Manipulation Will Always be Important. In Source Code Analysis and Manipulation (SCAM)* (pp. 7–19). IEEE Computer Society.

Hart, T. E., Ku, K., Gurfinkel, A., Chechik, M., & Lie, D. (2008). Augmenting Counterexample-Guided Abstraction Refinement with Proof Templates. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (pp. 387–390).

Haugh, E., & Bishop, M. (2003). *Testing C Programs for Buffer Overflow Vulnerabilities. Network and Distributed System Security (NDSS)*. The Internet Society.

Hentschel, M. (2016). *Integrating Symbolic Execution, Debugging and Verification* [Ph.D. dissertation]. Technische Universität Darmstadt.

Horstmann, C. S., & Cornell, G. (2005). *Core Java 2*. Palo Alto, CA: Sun Microsystems Press.

Ibing, A. (11 de 2014). A Backtracting Simbolic Execution Engine with Sound Path Merging. In *SECURWARE: The Eighth International Conference on Emerging Security Information, Systems and Technologies* (pp. 180-185).

Jackson, D., & Damon, C. A. (2000). Software Analysis: A Roadmap. In *International Conference on Software Engineering* (pp. 133-145). Limerick: ACM Press.

Jeffrey, D., Gupta, N., & Gupta, R. (2008). Identifying the root causes of memory bugs using corrupted memory location suppression. In *International Conference on Science and Technology of Synthetic Metals (ICSM)* (pp. 356-365). IEEE Computer Society. doi:10.1109/ICSM.2008.4658084

Johnson, B., Song, Y., Murphy-Hill, E. R., & Bowdidge, R. W. (2013). Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)* (pp. 672-681). IEEE Computer Society. doi:10.1109/ICSE.2013.6606613

Kim, Y., Lee, J., Han, H., & Choe, K.-M. (2010). Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology*, *52*(2), 210–219. doi:10.1016/j.infsof.2009.10.004

King, J. C. (1976). Symbolic Execution and Program Testing. *Commun. ACM, 19*, 385-394. doi:10.1145/360248.360252

Kitchenham, B., & Charters, S. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering.* Tech. rep., Keele University and Durham University Joint Report.

Larochelle, D., & Evans, D. (2001). Statically Detecting Likely Buffer Overflow Vulnerabilities. In *International Information Security Conference (SEC),* Washington, D.C.

Larson, E., & Austin, T. M. (2003). High Coverage Detection of Input-Related Security Faults. *USENIX Security Symposium.* USENIX Association.

Le, W., & Soffa, M. L. (2008). Marple: a demand-driven path-sensitive buffer overflow detector. In M. J. Harrold, & G. C. Murphy (Ed.), *SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (pp. 272-282). ACM. doi:10.1145/1453101.1453137

Le, W., & Soffa, M. L. (2011). Generating analyses for detecting faults in path segments. In M. B. Dwyer, & F. Tip (Ed.), *International Symposium on Software Testing and Analysis (ISSTA)* (pp. 320-330). ACM. doi:10.1145/2001420.2001459

Lhee, K., & Chapin, S. J. (2003). Buffer overflow and format string overflow vulnerabilities. *Software, Practice & Experience*, *33*(5), 423–460. doi:10.1002/spe.515

Li, B.-H., & Shieh, S. (2011). *RELEASE: Generating Exploits Using Loop-Aware Concolic Execution. In Secure Software Integration and Reliability Improvement (SSIRI)* (pp. 165–173). IEEE Computer Society.

Li, L., Cifuentes, C., & Keynes, N. (2010). Practical and Effective Symbolic Analysis for Buffer Overflow Detection. In *SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (pp. 317-326). New York, NY: ACM. doi:10.1145/1882291.1882338

Liang, B. (2009). Static Vulnerabilities Detection Based on Extended Vulnerability State Machine Model. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing* (Vol. 2, pp. 305-308). doi:10.1109/NSWCTC.2009.10

Livshits, V. B., & Lam, M. S. (2003). Tracking pointers with path and context sensitivity for bug detection in C programs. In *SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (pp. 317-326). ACM. doi:10.1145/940071.940114

Ma, K.-K., Yit Phang, K., Foster, J. S., & Hicks, M. (2011). *Directed Symbolic Execution*. Springer Berlin Haidelberg. doi:10.1007/978-3-642-23702-7_11

Meng, Q., Wen, S., Feng, C., & Tang, C. (2016). Predicting buffer overflow using semi-supervised learning. In *International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)* (pp. 1959-1963). IEEE. doi:10.1109/CISP-BMEI.2016.7853039

MITRE. (2017). Common Weakness Enumeration (CWE)---a community-developed list of software weakness type. Retrieved from https://cwe.mitre.org/

Mouzarani, M., Sadeghiyan, B., & Zolfaghari, M. (2015). A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes. In *Pacific Rim International Symposium on Dependable Computing (PRDC)* (pp. 42-49). IEEE Computer Society. doi:10.1109/PRDC.2015.10

Mouzarani, M., Sadeghiyan, B., & Zolfaghari, M. (2016). Smart fuzzing method for detecting stack-based buffer overflow in binary codes. *IET Software*, *10*(4), 96–107. doi:10.1049/iet-sen.2015.0039

Muntean, P., Kommanapalli, V., Ibing, A., & Eckert, C. (2015). Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT. *Computer Safety, Reliability, & Security*, *9337*, 441–456.

Nagarakatte, S., Zhao, J., Martin, M. M., & Zdancewic, S. (2009). SoftBound: Highly compatible and complete spatial memory safety for C. *ACM SIGPLAN Notices*, *44*(6), 245–258. doi:10.1145/1542476.1542504

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of Program Analysis*. Springer. doi:10.1007/978-3-662-03811-6

Novark, G., Berger, E. D., & Zorn, B. G. (2007). *Exterminator: automatically correcting memory errors with high probability. In Programming Language Design and Implementation (PLDI)* (pp. 1–11). ACM.

OWASP. (2016). Mobile Top 10 2016. Retrieved from https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10

Padaryan, V. A., Kaushan, V. V., & Fedotov, A. N. (2015). Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, *41*(6), 373–380. doi:10.1134/S0361768815060055

Padmanabhuni, B. M., & Tan, H. B. (2014). Predicting buffer overflow vulnerabilities through mining light-weight static code attributes. In *International Symposium on Software Reliability Engineering (ISSRE)* (pp. 317-322). IEEE Computer Society. doi:10.1109/ISSREW.2014.26

Padmanabhuni, B. M., & Tan, H. B. (2015a). Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *Computer Software and Applications Conference (COMPSAC)* (pp. 450-459). IEEE Computer Society. doi:10.1109/COMPSAC.2015.78

Padmanabhuni, B. M., & Tan, H. B. (2015b). *Light-Weight Rule-Based Test Case Generation for Detecting Buffer Overflow Vulnerabilities. In ICSE: Automation of Software Test (AST)* (pp. 48–52). IEEE Computer Society.

Padmanabhuni, B. M., & Tan, H. B. (2016). Auditing buffer overflow vulnerabilities using hybrid static-dynamic analysis. *IET Software*, *10*(2), 54–61. doi:10.1049/iet-sen.2014.0185

Pozza, D., & Sisto, R. (2008). A Lightweight Security Analyzer inside GCC. In *International Conference on Availability, Reliability and Security (ARES)* (pp. 851-858). IEEE Computer Society.

Pozza, D., Sisto, R., Durante, L., & Valenzano, A. (2006). *Comparing lexical analysis tools for buffer overflow detection in network software. In Communication Systems Software and Middleware (COMSWARE)* (pp. 1–7). IEEE.

Rawat, S., & Mounier, L. (10 de 2010). An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A Case of Aiming in Dim Light. In *European Conference on Computer Network Defense (EC2ND)* (pp. 37-45). doi:10.1109/EC2ND.2010.14

Rawat, S., & Mounier, L. (2012). *Finding Buffer Overflow Inducing Loops in Binary Executables. In Software Security and Reliability (SERE)* (pp. 177–186). IEEE.

Sadeghi, A., Bagheri, H., Garcia, J., & Malek, S. (2017). A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android apps. *IEEE Transactions on Software Engineering*, *43*(6), 492–530. doi:10.1109/TSE.2016.2615307

Sebesta, R. (2012). *Concepts of Programming Languages* (10th ed.). Person.

Shahriar, H., & Zulkernine, M. (2008). Mutation-Based Testing of Buffer Overflow Vulnerabilities. In *Computer Software and Applications Conference (COMPSAC)* (pp. 979-984). IEEE Computer Society.

Shahriar, H., & Zulkernine, M. (2011). *A Fuzzy Logic-Based Buffer Overflow Vulnerability Auditor. In Dependable, Autonomic and Secure Computing (DASC)* (pp. 137–144). IEEE Computer Society.

Shaw, A., Doggett, D., & Hafiz, M. (2014). *Automatically Fixing C Buffer Overflows Using Program Transformations. In Dependable Systems and Networks (DSN)* (pp. 124–135). IEEE Computer Society.

Skerret, I. (2017). IoT Developer Trends 2017 Edition. Retrieved from https://ianskerrett.wordpress.com/2017/04/19/iot-developer-trends-2017-edition/

Spafford, E. H. (2003). A Failure to Learn from the Past. In *Annual Computer Security Applications Conference (ACSAC)* (pp. 217-231). IEEE Computer Society. doi:10.1109/CSAC.2003.1254327

Teixeira, F. A., Machado, G. V., Pereira, F. M., Wong, H. C., Nogueira, J. M., & Oliveira, L. B. (2015). *SIoT: securing the internet of things through distributed system analysis. In Information Processing in Sensor Networks (IPSN)* (pp. 310–321). ACM. doi:10.1145/2737095.2737097

Tonella, P. (7 de 2004). Evolutionary Testing of Classes. *SIGSOFT Softw. Eng. Notes, 29*, 119-128. doi:10.1145/1013886.1007528

Viega, J., Bloch, J. T., Kohno, Y., & McGraw, G. (2000). ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society. doi:10.1109/ACSAC.2000.898880

Vujosevic-Janicic, M. (2008). Ensuring Safe Usage of Buffers in Programming Language C. In *International Conference on Software Technologies (ICSOFT)* (pp. 29-36). INSTICC Press.

Wagner, D., Foster, J., Brewer, E., & Aiken, A. (2000). A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. Network and Distributed Systems Security Conference* (pp. 3-17). ACM Press.

Weber, M., Shah, V., & Ren, C. (2001). *A Case Study in Detecting Software Security Vulnerabilities Using Constraint Optimization. In Source Code Analysis and Manipulation (SCAM)* (pp. 3–13). IEEE Computer Society.

Weiser, M. (7 de 1984). Program Slicing. *IEEE Computer Society Trans. Software Engineering, 10*, 352-357.

Wheeler, D. A. (2018). Flawfinder. Retrieved from https://www.dwheeler.com/flawfinder/

Wilander, J., & Kamkar, M. (2003). *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. Network and Distributed System Security (NDSS)*. The Internet Society.

Woodraska, D., Sanford, M., & Xu, D. (2011). Security mutation testing of the FileZilla FTP server. In *Symposium On Applied Computing (SAC)* (pp. 1425-1430). ACM. doi:10.1145/1982185.1982493

Wu, Y., Siy, H. P., & Gandhi, R. A. (2011). Empirical results on the study of software vulnerabilities. In *International Conference on Software Engineering (ICSE)* (pp. 964-967). ACM. doi:10.1145/1985793.1985960

Xu, R.-G., Godefroid, P., & Majumdar, R. (2008). Testing for Buffer Overflows with Length Abstraction. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.1145/1390630.1390636

Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and Discovering Vulnerabilities with Code Property Graphs. In *IEEE Symposium on Security and Privacy* (pp. 590-604). IEEE Computer Society. doi:10.1109/SP.2014.44

Yan, K., Liu, D., & Meng, F. (2015). A highly automated binary software vulnerability risk evaluation method for off-by-one stack based buffer overflow. In *2015 IEEE International Conference on Computer and Communications (ICCC;* pp. 16-20). doi:10.1109/CompComm.2015.7387532

Ye, T., Zhang, L., Wang, L., & Li, X. (2016). An Empirical Study on Detecting and Fixing Buffer Overflow Bugs. In *International Conference on Software Testing (ICST)* (pp. 91-101). IEEE Computer Society. doi:10.1109/ICST.2016.21

Younan, Y., Joosen, W., & Piessens, F. (2012). Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Comput. Surv., 44*, 17:1-17:28. doi:10.1145/2187671.2187679

Zeller, A. (2009). *Why Programs Fail - A Guide to Systematic Debugging* (2nd ed.). Academic Press.

Zhang, C., Wang, T., Wei, T., Chen, Y., & Zou, W. (2010). IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In *European Symposium on Research in Computer Security (ESORICS)* (pp. 71-86). Springer. doi:10.1007/978-3-642-15497-3_5

Zhang, C., Zou, W., Wang, T., Chen, Y., & Wei, T. (2011). Using type analysis in compiler to mitigate integer-overflow-to-buffer-overflow threat. *Journal of Computer Security*, *19*(6), 1083–1107. doi:10.3233/JCS-2011-0434

ENDNOTES

[1] OWASP – Open Web Application Security Project (https://www.owasp.org)
[2] https://www.scopus.com
[3] https://webofknowledge.com
[4] The final search strings can be found at: https://goo.gl/oo1ez1
[5] Sub-programs in the C jargon
[6] Unsigned integer variables hold only positive numbers whereas signed integer variables positives and negatives integers.Marcos Lordello Chaim obtained BSc (1987), MSc (1991) and PhD (2001) degrees in Electrical Engineering from the State University of Campinas, Campinas, Brazil. From 1990 to 1992, he was assistant professor at São Paulo State University (UNESP), São José do Rio Preto, Brazil. After leaving UNESP, he worked at Embrapa Information Technology where he developed software for the agribusiness domain until 2005. Since then, he is faculty member of the School of Arts, Sciences and Humanities (EACH), University of São Paulo (USP), São Paulo, Brazil. His research interests lie on the following topics: software testing, debugging, and maintenance; development methods; empirical software engineering; and information systems education.

Daniel Soares Santos is a Ph.D. candidate in Computer Science and Computational Mathematics at the University of São Paulo (USP), Brazil. He received his BS degree in Computer Science from State University of Southwestern Bahia (UESB), and his MS degree in Computer Science from USP with an internship abroad at Fraunhofer-IESE, Germany. His main research interests include software architecture, systems-of-systems, big data, machine learning, and software quality assurance.