CorCA: An Automatic Program Repair Tool for Checking and Removing Effectively C Flaws

João Inácio, Ibéria Medeiros
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
joaommtinacio@gmail.com, imedeiros@di.fc.ul.pt

***Abstract*—Embedded systems are present in many devices, such as the Internet of Things, drones, and cyber-physical systems. The software security of these devices can be critical, depending on the context they are integrated and the role they play (e.g., water plants, vehicles). C is the core language used to develop the software for these devices and is known for missing the bounds of its data types, which leads to vulnerabilities such as buffer overflows. These vulnerabilities, when exploited, can cause severe damage and put human life in danger. One of the concerns with vulnerable C programs is to correct the code automatically and adequately, employing secure code that can remove the existing vulnerabilities and avoid attacks. However, such a task faces some challenges, namely determining what code is needed to remove them and, at the same time, ensuring the correct behaviour of the program, where to insert it, and verifying that the correction applied is secure and effectively removes the vulnerabilities. Another challenge is to accomplish all these elements in an automated manner. This paper presents an approach that automatically, after discovering and confirming potential vulnerabilities of an application, applies code correction to fix the vulnerable code of those confirmed vulnerabilities and validates the new code. We implemented the approach, resulting in the CorCA [1] tool, and evaluated it with a set of tests and real applications. The experimental results showed that the tool was capable of detecting vulnerabilities and fixing them correctly.**

***Index Terms*—Code Repair, Buffer Overflow Vulnerabilities, Static Analysis, Fuzzing, Software Security**

I. Introduction

The advancement of technologies and the growth in the use of software systems daily and globally have raised several questions related to the security of the software used. In our everyday life, we use several devices (e.g., smartphones, computers, vehicles) whose operation depends on the software they use. These devices are in constant development and evolution, always searching for bringing new features and a better user experience, and their software has become more robust and complex to provide such features. The increase in complexity and size favours the appearance of bugs in code since it becomes harder to analyze and ensure its correctness. Under certain conditions to which systems are submitted, these bugs can cause the appearance of exploitable vulnerabilities leading to the corruption of the systems.

The existence of bugs in systems occurs due to the usage of unsafe languages and unintentional errors introduced by programmers. Although today there is a concern with software security, unsafe languages are still widely used, and errors continue to be made and are one of the main problems in building secure systems. C is one of the unsafe and most used languages in the development of software products in several areas. Even with the appearance of new languages, it remains one of the most used [2]. At the same time, C lacks checking mechanisms, such as buffer limits, leaving the developer entirely responsible for the correct memory and resource management. These weaknesses are at the root of buffer overflows (BO) vulnerabilities, which range the first place in the CWE's top 25 of the most dangerous weaknesses [3]. The exploitation of BO when existing in critical safety systems, such as railways and autonomous cars, can have catastrophic effects on manufacturers or endanger human lives. However, C contains safe functions that can be used to avoid introducing vulnerabilities and invalidating attacks. But, developers may not be aware of these functions or even know how to use them properly.

Currently, there is a great demand for tools that help the development of secure software to overcome the aforementioned difficulties. However, such tools can be hard to use and can report vulnerabilities that are not real, i.e., false positives. For this reason, many tools require developers to manually analyze the reported results, which consumes a significant amount of developers' time. Moreover, this time is ineffective when they look for inexistent vulnerabilities in the source code. These tools can use different techniques to detect vulnerabilities, being fuzzing the most used for its ability to exploit them [4] [5] [6]. But, fuzzing does not give information about them on the code, putting this task on the programmers' side, which can be challenging for those who do not know about security programming. Static analysis [7] [8], the combination of it with fuzzing [9] [10], and recently machine learning approaches [11] [12] have been proposed to identify bugs in the code, but they suffer from imprecision, putting once again the effort of checking their output veracity on developer's side.

Hence, it is necessary to find ways to automatically detect flaws and remove them by employing more security programming to be helpful for developers. The existence of tools capable of automatically detecting and fixing vulnerabilities would make developers' tasks easier and decrease the time needed to write secure code. For automatic program repair (APR) for C there are few tools available with these capabilities [13][14][15][16][17] [18], and they have some limitations, such as producing syntactically incorrect code or not verifying the effectiveness of the inserted fixes [15][16]. Moreover, most of them are not for security and the existing ones do not verify the correctness of the fixed code, which can leave the programs syntactically incorrect. Hence, it is a must to carry out tools to remove vulnerabilities in C programs by correcting their code, making it safe. Furthermore, it is necessary, on the one hand, to confirm the existence of the vulnerabilities found for the reduction of false positives and, on the other hand, to verify the correctness and effectiveness of the corrections made.

This paper proposes an approach for automatically detecting and correcting BO vulnerabilities in C programs. The idea behind the approach is to combine techniques of static analysis, fuzzing and APR to discover BOs statically, confirm their presence by fuzzing and remove the vulnerabilities by repairing the code and testing the corrections' effectiveness.

The paper also presents the *Correction C Automatically* (`CorCA`) tool that implements the approach. `CorCA` first hits all sensitive sinks associated with BO (e.g., `strcpy`) existing in the program under testing (PUT), next extracts the code slice (a single data flow that starts at a buffer declaration and ends at a sensitive sink) for each function hinted and composes its slice program syntactically correct and executable. Afterwards, it fuzzes the slice programs to confirm which ones are really vulnerable and then corrects them by applying fixes (small pieces of code). Fixes are generated and inserted automatically and contain the right code needed to remove the vulnerabilities. Next, the fixed slice programs are compiled and submitted to a validation process, where they are fuzzed with the test cases that exploited the vulnerabilities contained therein and with new test cases driven from the former. Lastly, a new release of the PUT is produced with the validated fixes. The tool was validated with 1075 programs from SARD [19] and assessed over 7 real applications, where discovered *6 zero-days vulnerabilities* (i.e., 6 previously unknown vulnerabilities) and correctly fixed the vulnerable applications.

The main contributions of the paper are: (1) an approach for searching for potential BO vulnerabilities based on static analysis and their confirmation through the generation of test cases derived from fuzzing, and the automatic creation of fixes to remove the vulnerabilities, their application, and assessment of their effectiveness; (2) a tool capable of flagging and confirming BO vulnerabilities in programs written in C, correcting them, and verifying the effectiveness of the corrections in an automated way; (3) an experimental evaluation that shows the ability of this tool to detect known and zero-day vulnerabilities and remove them effectively.

II. Background and Related Work

This section presents the background on BO vulnerabilities, the techniques of vulnerability detection, focusing on those related to our work, and a review of techniques for APR.

A. Vulnerabilities

A vulnerability can be described as a flaw or weakness in a system, which can be exploited or triggered by a threat source resulting in a security breach or a violation of the system's security policy [20][21][22]. Vulnerabilities usually result from introduced flaws in the software during its development. C is a very flexible language that facilitates access to memory in an invalid and unchecked manner. These characteristics lead to many security flaws because programmers assume that the language handles certain aspects when, in fact, it does not.

Buffer overflows, in C programs, are the root of a large percentage of severe security problems that have emerged over the years [23][24][25][26]. They can be expressed in code and exploited in several manners [27], although there are various countermeasures that can be implemented to prevent them [28]. A BO occurs when a program performs operations outside of the boundaries of the memory allocated to a particular buffer (a contiguous chunk of memory space of the same data type). The root cause of most BOs is the combination of memory manipulation and wrong assumptions about the size or composition of data. They usually involve violating the assumptions made by the programmers when using memory manipulation functions (e.g., `gets`, `strcpy`) that do not perform bounds checking of the buffers on which they operate. Even bounded functions, the well-known safe functions such as `strncpy`, can cause BOs when used incorrectly. Depending on the size of the overflow and the memory location, a BO can go unnoticed but can corrupt data, cause erratic behaviour, cause the execution of malicious code, or terminate the program abnormally.

The code in Listing 1 illustrates a simple BO. The code uses the `gets` function to read an arbitrary amount of data into a buffer. The safety of the code depends on the user to always enter fewer characters than `BUFSIZE` because there is no way to limit the amount of data read by this function.

```
#include <stdio.h>
#define BUFSIZE 20
int main(int argc, char **argv[]) {
  char buffer[BUFSIZE];
  gets(buffer);
  return(0);
}
```

Listing 1: Buffer overflow example.

B. Vulnerability Detection

The correct vulnerability identification is a hard task that requires a considerable amount of time. The creation of automatic methods to obtain tools that accomplish this task easier and faster has emerged in the last years. These tools have employed various techniques, such as static and dynamic analysis, fuzzing, and recently machine learning (ML).

Static analysis tools aim to find bugs in the code of an application through the analysis of its code and without executing it [29][30][31][32]. Opposingly dynamic code analysis tools inspect programs while running to identify potential issues that arise during the actual execution of the program and impact its reliability. [33][34][35]. Fuzzing is a popular software testing method, mainly used for security testing, that injects random inputs into a system to reveal software defects and vulnerabilities by monitoring the system for exceptions such as crashes or information leakage [36][37]. There is a great diversity of work related to fuzzers and the different types of fuzzers, from mathematical models, solving constraints, white- and grey-box, and combinations of different techniques, e.g., static and dynamic analysis [38][39][40][41][42][43][44][45][46][47]. In addition, some research seeks to use machine learning techniques to find vulnerabilities in the code, taking advantage of models to predict where vulnerabilities may exist and thus make the detection process faster [48][49][50][51][52].

C. Automatic Program Repair

The repair techniques follow two main approaches: software healing (or state repair) and software repair (or behavioural repair) [53]. The former consists of changing the state (e.g., stack, heap) of the program under repair (PUT) through a healing process composed of two steps that might be executed iteratively: (1) the healing step executes a healing operation that can prevent or mitigate a failure detected; (2) the verification step checks if the program runs as expected after the healing operation has finished [54]. The second approach changes the program's behaviour by altering the code, which can be done offline or at runtime. This process comprises three steps that might be executed iteratively: (1) the localization step identifies the locations in the code where a fix could be applied; (2) the fix step generates the fixes that will modify the code in the locations returned by (1); (3) the verification step checks if the fixes have really repaired the program.

Most existing works in the literature follow the second approach with the additional step of code instrumentalization to discover BOs and the places into the code to insert the fixes [14][13]. Some others do not apply the fix verification step of this approach [16][55][56]. Yet others only suggest how the vulnerabilities found can be repaired [15][57]. The approach we propose follows the three steps of the software repair approach without requiring instrumentalising the code as it resorts to static analysis to identify potential vulnerabilities and gather the code involved in these to produce executable slice programs containing the data flow paths of each potential vulnerability. Also, the approach uses fuzzing to exercise the resulting slice programs.

Recently, advancements in machine learning brought a wave of progress within the field of software repair. This trend leads to the appearance of different repair techniques named data-driven approaches that make decisions based on analysis and interpretation of hard data rather than on observation [58][59][60][61][62][63].

III. Challenges

We aim to create an automated BO vulnerability discovery and fixing, and code correction validation tool. We present next the challenges it should address and the key ideas that emerged to cope with them and their reasoning.

A. How to find vulnerabilities and ensure they are exploitable?

One of the problems related to the identification of vulnerabilities in code through static analysis tools is the production of false positives (FP), which it turns difficult to locate real vulnerabilities in the source code. Hence, it is necessary to ensure that the vulnerabilities found statically are exploitable, giving evidence of such by providing exploits (test cases).

Our idea to overcome this challenge is to analyze the code of a program through static analysis to discover candidate vulnerabilities, whose results can contain FPs. Next, we will use fuzzing to filter these results, checking when it exploits the candidate vulnerabilities, producing their exploits.

B. How to generate compilable and executable programs from code slices of vulnerabilities found statically?

Fuzzing to exercise the PUT demands that the program is running, so an executable file of it is required. On the other hand, static analysis tools neither provide executable nor compilable code for the flaws they report. Most of them only output the line of code of the sensitive sink and, some others, which entry point supposedly hit it. Very few tools return the complete slice of the vulnerable code, i.e., the lines of the vulnerable execution path that starts at an entry point and ends at a sink, but this code is neither compilable nor executable. Thus, the challenge here is how to capture all the code needed for each vulnerability reported by static analysis and make it executable to be exercised by fuzzing.

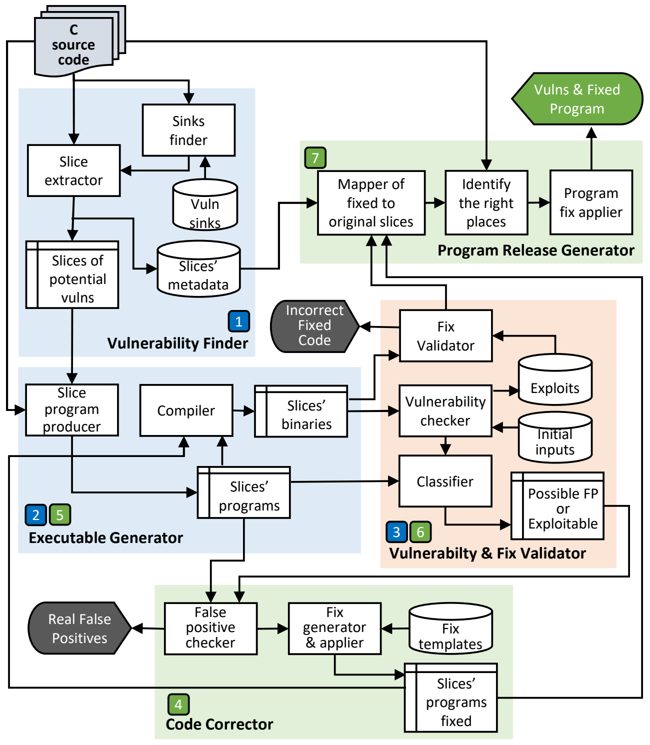
The main idea is to generate a *slice program* for each potential vulnerability found, composed of the slice of the vulnerable code and all the other code necessary to make it compilable and executable (e.g., the main function). For that, we will employ code parsing techniques over the static analysis output and the PUT files to capture all needed code.

C. Where and how to correct the code?

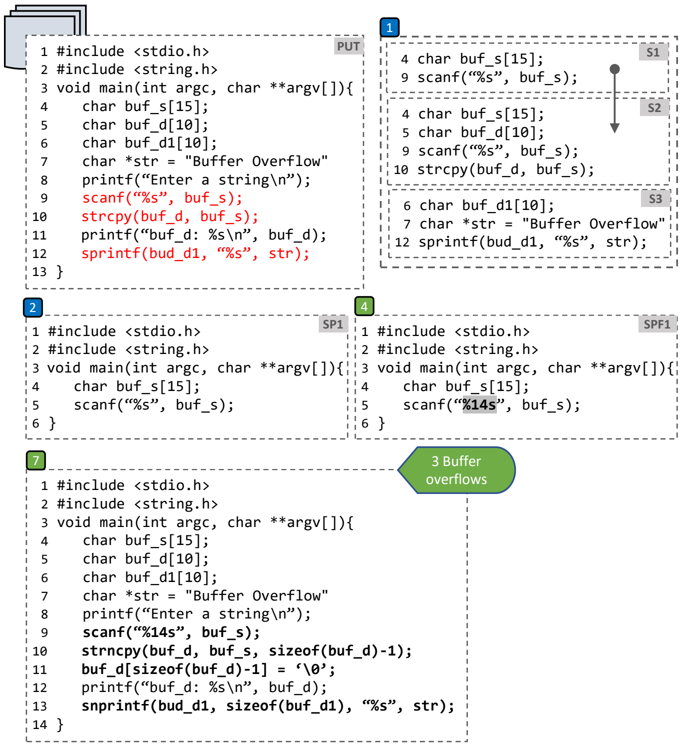
One of the main challenges of automatic code correction is to decide where the code for removing the vulnerability – *fix* – should be inserted. This decision is difficult because a slight change in the code may alter the program's logic, and its effect needs to be avoided to maintain the correct program's behaviour. As we want to fix vulnerabilities associated with BOs, which are usually related to sensitive sinks, our focus will be on fixing the lines of these sinks. Hence, we propose the inclusion of fixes in these lines or close to them.

Another challenge associated with code correction is to decide what type of correction to apply in each case, since a given vulnerability class can be expressed in the code in different forms, even for the same sensitive sink. Moreover, as there is no universal fix for all cases and each sensitive sink is used differently with distinct arguments, it makes the process of correction more difficult. Our idea is to fix the issues associated with specific sensitive sinks. For some sensitive sinks, the fix may be to replace them with their secure version (e.g., `strncpy` for `strcpy`), but some may have no possibility of doing this because they do not have a safe version (e.g., `scanf`). But in both cases, our approach will capture the sinks' arguments and whether they need some validation before using them in the sinks, and determine the correct amount of bytes that must be used by the fixes.

In sum, our conception to solve these two challenges is to construct a set of fix templates that will be used dynamically



(a) Overview of the proposed approach.



```
#include <stdio.h>
#include <string.h>
void main(int argc, char **argv[]){
   char buf_s[15];
   char buf_d[10];
   char buf_d1[10];
   char *str = "Buffer Overflow"
   printf("Enter a string\n");
   scanf("%s", buf_s);
   strcpy(buf_d, buf_s);
   printf("buf_d: %s\n", buf_d);
   sprintf(bud_d1, "%s", str);
}
```

```
 4 char buf_s[15];
 9 scanf("%s", buf_s);
```

```
 4 char buf_s[15];
 5 char buf_d[10];
 9 scanf("%s", buf_s);
10 strcpy(buf_d, buf_s);
```

```
 6 char buf_d1[10];
 7 char *str = "Buffer Overflow"
12 sprintf(bud_d1, "%s", str);
```

```
#include <stdio.h>
#include <string.h>
void main(int argc, char **argv[]){
   char buf_s[15];
   scanf("%s", buf_s);
}
```

```
#include <stdio.h>
#include <string.h>
void main(int argc, char **argv[]){
   char buf_s[15];
   scanf("%14s", buf_s);
}
```

```
#include <stdio.h>
#include <string.h>
void main(int argc, char **argv[]){
   char buf_s[15];
   char buf_d[10];
   char buf_d1[10];
   char *str = "Buffer Overflow"
   printf("Enter a string\n");
   scanf("%14s", buf_s);
   strncpy(buf_d, buf_s, sizeof(buf_d)-1);
   buf_d[sizeof(buf_d)-1] = '\0';
   printf("buf_d: %s\n", buf_d);
   snprintf(bud_d1, sizeof(buf_d1), "%s", str);
}
```

(b) Example of execution of the approach.

Fig. 1: Approach architecture overview and execution example.

when the vulnerable code is being inspected to determine what is necessary to fix and where the fix will be inserted. Based on these inspections, the fix template is selected and generated the final fix. Some fixes will replace sensitive sinks with their safer version if they exist. Otherwise, fixes will modify the sink statement or insert code instructions close to it to ensure that the sensitive sink is used correctly and safely.

D. How to determine that the fix applied is effective?

We propose to automate this validation process by using the exploits generated in the fuzzing task during the vulnerability confirmation step (i.e., the exploits that break the functioning of the PUT) to verify that the fix works. If the exploits cannot crash the program's operation, the applied fix effectively removed the vulnerability and corrected the code. Also, if they cannot hang the program's behaviour and logic, the fix was correctly generated syntactically and inserted in the right places in the code. In addition, our validation process will generate new test cases to try to circumvent the fix and spoil the program's behaviour.

IV. Design Insights and Approach

A. Approach Overview

We present an approach that identifies and fixes BO vulnerabilities in the source code of C programs and verifies the effectiveness and correctness of the corrected (fixed) code in an automated manner. To pursue this goal and cope with the challenges of Section III, the approach employs static analysis to find possible BO vulnerabilities, fuzzing to confirm the BO found and validate the effectiveness of the code fixed, and code repair to correct the code automatically with fixes generated dynamically based on fix templates.

Fig. 1(a) shows an overview of the approach architecture with their five modules, divided into the three phases of the approach: *detection*, *fixing* and *validation*. The first two phases correspond to the blue and green boxes in the figure and employ source code static analysis to detect and remove vulnerabilities and generate a new version of the PUT. The validation phase (the orange box) operates in runtime to confirm the existence of the previously flagged vulnerabilities and validate the code correction made and its effectiveness.

B. Vulnerability Finder

The first fundamental task of our approach is to locate the potential vulnerabilities, and the Vulnerability Finder is the module that performs that action. Through static analysis techniques, it analysis the code of the program for searching for potential vulnerabilities related to sensitive sinks associated with BO, collects information about the vulnerabilities and their location in the program, and extracts their code slices.

Considering that BOs are associated with missing input validation or bound checking before data/memory manipulation or with calling functions that may overwrite the allocated bounds of buffers, a slice is a *single data flow* that starts at an entry point and ends at a sensitive sink. Between the entry point and the sensitive sink, the slice contains all instructions and variables dependent on them. However, an entry point can be an input-sensitive sink (e.g., `scanf` and `gets`) since the data

TABLE I: Functions associated with buffer overflows stratified by categories, and their safe versions.

Category	Vulnerable	Safe version	How the function handles the null character (\0)	Example
Input	gets	fgets	indicates N, reads N-1 bytes, adds \0 to N byte	char buf[8]; fgets(buf, stdin, 8);
	scanf	–	indicates N-1 bytes, reads N-1 bytes, adds \0 to N byte	char buf[8]; scanf(”%7s”, buf);
	sscanf	–		
	fscanf	–		
	vscanf	–		
	vsscanf	–		
	vfscanf	–		
Output	sprintf	snprintf	indicates N, writes N-1 bytes, adds \0 to N byte	char buf[8]; char *str=”Buffer overflow”; snprinf(buf, sizeof(buf), ”%s”, str);
	vsprintf	vsnprintf	indicates N, writes N-1 bytes, adds \0 to N byte	
Data manipulation	strcpy	strncpy	indicates N-1, writes N-1 bytes, we must add \0 to N byte	char buf[8]; char *str=”Buffer overflow”; strncpy(buf, str, sizeof(buf)-1); buf[sizeof(buf)-1]='\0';
	strcat	strncat	indicates N-1, writes N-1 bytes, adds \0 to N byte	char buf[10]; char *str1=”Buffer”; char *str2=” overflow”; strncpy(buf, str1, strlen(str1)); strncat(buf, str2, sizeof(buf)-strlen(str1)-1);
Memory manipulation	memcpy	–	indicates N, writes N bytes, adds \0 if the N byte is \0 OR	char buf[8]; char *str=”Buffer”; memcpy(buf, src, strlen(str)+1);
	memmove	–	indicates N-1, writes N-1 bytes, we must add \0 to N byte	
	memset	–	if the N byte is not \0	

read through that sink can overflow the buffer destination. Table I, column 2, presents the functions we considered and their stratification by four categories. *Input* and *output* categories allocate the functions regarding read and print data to a buffer, respectively. *Data* and *memory manipulation* categories contain the functions that allow copying, moving, or concatenating data/memory to a buffer or between buffers.

The vulnerability finder employs two steps – *sinks finder* and *slice extractor*. First, it scans the source code of the PUT, looking for sensitive sinks and outputting a list of hits that contain the potential vulnerabilities and their location in the code, i.e., the line of the code where the sinks' instructions are. In the second step, for each hit, it extracts the slice containing all instructions associated with it. It starts by analyzing the sink instruction and collects information about the variables used in the sink. Next, it parses the PUT's code to gather the code lines associated with these variables. To do so, after the source code is parsed, the slice extractor performs a bottom-up approach for tracking the variables and the ones dependent on them to where they are declared and initialized, and then extracts all lines associated with them. The resulting lines of code are combined with the sensitive sinks' line, generating a slice, which will be forwarded to the next module.

Fig. 1(b) shows an example of the execution of the approach. At its top-left, it is illustrated the code of a hypothetical program to be analyzed (PUT) that contains three sensitive sinks discovered by the sinks finder (lines 9, 10, and 12). For instance, when this code is analyzed, the sink finder outputs line 9 as being a potential BO. Next, the slice extractor obtains the `buf_s` variable argument from the `scanf` sink and then goes up along the code until finding line 4, where the variable is declared. At the end of this process, these two lines are combined, generating thus the slice `S1`. The top-right of the figure presents the three slices (`S1` to `S3`) extracted.

Alongside the slice extraction, the data collected about each slice is stored in a data structure constituted by three parts, as illustrated in Fig. 2. The first part contains general information about the slice itself: if it is included and/or contains another one, it is vulnerable, and its fix will be applied in the fixed and new version of the PUT. The second part is filled with the

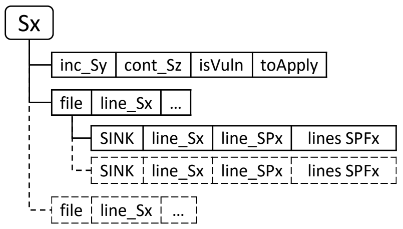


Fig. 2: Data structure for representing slices.

locations (file and code lines) from where it was extracted from PUT. Lastly, for each sink, the lines of code where it appears along the approach's pipeline are kept in the structure (explain next). This data structure is filled throughout the approach execution by the modules. For example, the left side of Fig. 3 shows the beginning of filling this structure for `S1` to `S3` slices, after they are extracted, and considering the code of Fig. 1(b) is in the file *put.c*. For `S1`, it is visible that it is included in `S2`, the lines extracted from PUT were 4 and 9, and the sink `scanf` is in line 9. For `S2`, it is stored that it contains `S1` and has two sinks in lines 9 and 10.

The first part of the data structure when filled, the slice extractor module also records the relationships between the slices. After all the slices are composed, this slices' connection information is used to define the order that slices will be processed along the pipeline. Slices without dependencies (e.g., `S3`) and slices that are part of others, but do not contain

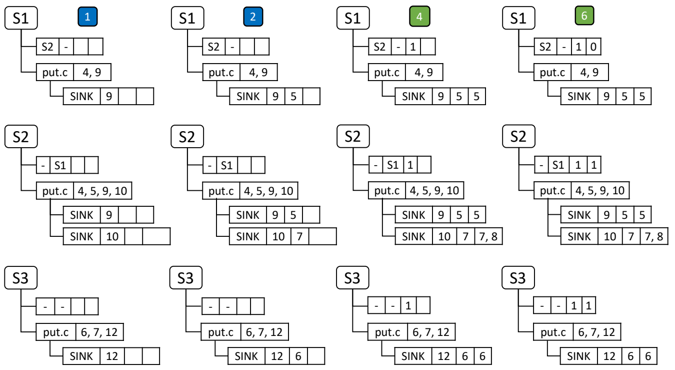


Fig. 3: Data structure filled along the approach's pipeline.

others (e.g., `S1`), should be processed first and in parallel. The remaining slices (those that include the former) are processed next. An example of this order for Fig. 1(b) could be `S1`, `S3` and `S2`. This order is important because we ensure that exploits of slices with subsequent connections are available to the correction validation phase of slices with precedence.

C. Executable Generator

The extracted slices that potentially contain a vulnerability and which we intend to correct, their code is neither compilable nor executable. On the other hand, fuzzing requires that the programs to be fuzzed must be executable. To satisfy this requirement, the goal of the executable generator module is to create for each slice a complete program syntactically correct and compilable, that we denominate by *slice programs*. The module comprises two submodules – *slice program producer* and *compiler* – to cope with this goal.

1) Slice Program Producer: Produces the slice programs of the slices it receives from the vulnerability finder. It parses the code slices to collect information about the sensitive sinks and variables used. Next, it statically analyzes the code of the original program to extract other necessary code instructions related to constants, directives, and functions, intending to get a functional program file. Once this data is obtained, it is added to the slices containing, thus, the lines required to turn them accordingly to the language's syntax. After this task, the resulting file contains a `main` function to be executed, the necessary libraries, and the slices with all the required information for the file to be compiled. For libraries, we chose to include all those that the parser found throughout the PUT code, as the goal is to produce compilable programs regardless of including more libraries than they need. In addition, it is registered the sink's line numbers of the resulting slice program, which later will be used by the *Program Release Generator* (see Section IV-G).

The left-middle of Fig. 1(b) illustrates the slice program (`SP1`) generated for the slice `S1`, where the main function and the libraries were included in `S1` to produce a compilable program. The second column of Fig. 3 shows the data structure of each slice updated with the sink line on `SP`. For instance, for `S1`, line 5 of `SP1` was added to the `S1`' data structure, meaning that the sink `scanf` in `SP1` is located in line 5.

2) Compiler: The compiler submodule generates the executable of each slice program, for then be used by the Vulnerability & Fix Validator module. It works in two distinct phases in our approach, since that module performs two distinct validation tasks with two versions of the slice program under test at different moments. The next section details these validation phases. Although the Compiler works in distinct phases, the tasks it performs are the same. It compiles the produced programs according to the requirements for using the fuzzing technique in the validation process. However, in the first phase, the program it compiles contains the potential vulnerability we want to test and fix, whereas in the second phase, the file it receives is the one that fixes the vulnerability, which we intend to validate in runtime.

D. Vulnerability & Fix Validator

This module operates in two distinct phases, respectively, for validating the existence of vulnerabilities and the effectiveness of the code correction. The first is to check whether the potential vulnerabilities found by the Vulnerability Finder are real or not. The second is to check whether the fixes applied by the Code Corrector are effective or not. It uses the fuzzing technique to pursue both validations, following the processing order of slices previously established in the slice extraction step (see Section II-B), and comprises three submodules: *Vulnerability Checker*, *Classifier* and *Fix Validator*. For a better understanding of the approach's pipeline, the last submodule is described after the Code Corrector module.

Fuzzing is a well-known technique that can take a long time (hours or even days) to discover software flaws, as it generates inputs to simultaneously try to uncover code that has not yet been discovered and exploit any flaws in that code. However, both discoveries depend on various aspects, such as the complexity and control flow of the PUT, and the generated inputs. Keeping these issues in mind, the rationale behind extracting slices for each sink found statically, each containing a single data flow (i.e., a single execution path), is to cope with such issues. Hence, we propose an approach that exercises slices with fuzzing, rather than all the PUT code, for a faster and easier way to attempt to exploit vulnerabilities.

1) Vulnerability Checker: Performs the first validation phase, i.e., the exploitation of the potential vulnerabilities. To proceed with this task, it fuzzes the executable slice program with inputs produced by mutation, namely by bit flips (sequential bit flips of varying lengths and step-overs) and arithmetic (addition and subtraction of small integers) operations. The fuzzing starts with a standard and benign input (e.g., a string) to trigger the loop of input mutation, which will generate inputs, by mutating the previous, and test them with the slice program. As a slice only contains a single data flow, it is expected that its exploitation does not take much time. Hence, the fuzzing process is active for a given short period of time (e.g., one minute) for trying to produce an input capable of exploiting the possible vulnerability under test in a fast way. The successful inputs we call exploits and they are stored to be used later, in the second validation phase.

2) Classifier: The potential vulnerabilities not exploited during the fuzzing period are marked as *possible false positives* (PFP), meaning that the vulnerability finder *probably* flagged a sink that does not empower a vulnerability or the fuzzer during the fuzzing period *might* not generate an input capable of exploiting the potential vulnerability. On the other hand, the potential vulnerabilities exploited are marked as such.

E. Code Corrector

This module performs the correction of vulnerabilities, and is divided into two submodules: *False Positive Checker* and *Fix Generator & Applier*.

1) False Positive Checker: This process starts by parsing the slice program received from the Validator (first phase) to locate the sensitive sinks associated with the vulnerabilities to be fixed, identify the variables used and register the sizes associated with them. Based on this information, it checks that the size of variables is in accordance with their use in sensitive sinks (e.g., the size of the buffer is correctly used in a sink) and determines whether the slices previously classified as PFP are real FP or vulnerabilities. Therefore, for slices marked as PFP that variable sizes are in concordance, they will be marked as real FP and reported as such, and no correction is made. Otherwise (i.e. the variable sizes do not agree), they proceed to correction, as it may happen that the time given for fuzzing may not have been enough to generate an exploit, and we want to invalidate the possibility of false negatives. On the other hand, slices flagged as exploitable will have their code fixed, regardless of the concordance between the sizes of the variables. Exploitable slices without concordance are real vulnerabilities. For the others, although they have at least an exploit, they will be fixed for prevention reasons, even if it is not necessary. After this decision is made, the field `isVuln` of the slice's data structure is filled with 1 (is vulnerable) or 0 (is not vulnerable or is a true FP). Fig. 3, third column, shows this field filled with 1, denoting that `S1` to `S3` are vulnerable.

2) Fix Generator and Applier: For the slices program market to suffer code correction, this submodule analyzes the information received from the previous module to understand what type of correction should be applied in each case.

Corrections are made in the sensitive sink instruction or close to it. For sinks that have a safe version (e.g., `strcpy`), your safe version (e.g., `strncpy`) will replace it. For those cases that do not have safe functions (e.g., `scanf`), their variables will be adjusted to the correct sizes. However, it may not be simple to use the safety functions, or even to make adjustments, autonomously. As BOs are linked to the number of bytes that will be written in a buffer, we must calculate the correct amount, taking into account how each safe function works and handles the null character (`\0`). In other words, when we specify `N` bytes in a function, we have to know whether this `N` in that function comprises the null character or not. For instance, in function `fgets` the specified `N` means that `N-1` bytes will be read and then the `N` byte is set to `\0`. But, in function `strncat` we have to specify `N-1` bytes to be concatenated, and the function sets `\0` to the `N` byte. A particular function is the `strncpy` that does not handle the null character, i.e., we must specify `N-1` bytes to be copied, and next add manually the `\0` to the byte `N`. Table I, columns 2 to 4, show the safe functions, how they handle the null character, and an example of how they can be used.

For generating the fixes, we defined a set of templates containing the instructions that correspond to the safe uses of the sensitive sinks with generic parameters. For each case, these generic parameters are modified by those specific to that case. For that, we resort to the information collected about the sink and the variable sizes to calculate the correct amount of bytes. For example, for `SP1` of Fig. 1(b), it would be collected the `scanf` sink, its parameter `"%s"` and `15` as size of `buf_s`. Next, the `scanf` template is parameterized with `14`; resulting the fix `scanf("%14s", buf_s);`. The `"%s"` argument was corrected to `"%14s"` to only be read 14 characters to the `buf_s` variable since this last can only store 15 bytes. The fifteen position of `buf_s` is reserved for `"\0"`, and hence we can only occupy the first 14 positions with data.

After fixes are generated, it is got the line number of the sink in the slice program (`SP`), replaces it with the fix and inserts new instructions if it is the case, producing thus *slice programs fixed* (`SPF`). In addition, the slice's data structure is updated with the line numbers of `SPF` that correspond to the ones on `SP` that were fixed and/or added. The middle-right of Fig. 1(b) illustrates the `SPF1` resulting from the correction of `SP1`, by fixing the function `scanf` at line 5. The third column of Fig. 3 shows the `SINK` updated with the line from `SPF1`. Also, we can observe for `S2` that line 7 of the `SP2` was fixed with lines 7 and 8 on `SPF2`, which correspond to the use of the `strncpy` function.

Finally, the resulting fixed slice programs are transmitted to the Compiler to be compiled and generate a new executable file and then forwarded to the Fix Validator.

F. Fix Validator

The second validation phase performed by the Vulnerability & Fix Validator module is to check whether the fix applied by the Code Corrector is effective. As in the first phase, it fuzzes the fixed slice program to try to break the code corrected, but this time it uses the previously stored exploits that exploited the vulnerability under processing. Furthermore, during the fuzzing process, these exploits are mutated in an attempt to discover new exploits that may break the applied fix. In this step, the validator uses the slice processing order list to also use the exploits of the slices preceding the one being analyzed and that their fixed slices were flagged to be applied for the fixed version of PUT (explained next). If all inputs (the stored exploits and the new ones) fail to break the fixed code, the applied fix is validated and considered effective. Otherwise, an alert is generated about the vulnerability found and that its correction needs the programmer's attention.

In addition, this submodule sets the `toApply` field to a value between 0 to 2, on the slice's data structure under analysis. It sets to 1 when the fix is considered effective and to 2 when the fix is broken. However, value 1 can change to 0 when the slice S_y under analysis is considered validated, but it contains a slice S_x where this field was set to 1 (we recall that precedent slices are tested first). In this way, for the generation of the new version of PUT, we only need to use S_y because it contains both valid corrections. In this case, the included slice S_x changes this field to 0 and the slice S_y keeps the value 1. The last column of Fig. 3 shows an example of this situation: `S1` is included in `S2` and both slices were correctly validated; the `toApply` field of `S1` is changed to 0, whereas `S2` keeps the 1 value; so, `S2` will be used in the PUT fixed version. Besides this case, another one can happen: the correction of slice S_y was broken. In such a case, S_y takes the value 2 and S_x keeps its value 1. Hence, just S_x will be used for the final correction.

G. Program Release Generator

When the second phase of the validation process (Section IV-F) ends without any exploit breaking the fixes, and it is found that the applied fix does not spoil the program's functioning, it means that the fix is effective and can be used to correct the original program. The Program Release Generator module is responsible for performing this correction, outputting a new release of the PUT with its files containing the corrected code, i.e., with the vulnerabilities fixed.

To perform the final correction, the module only uses the slices that their `toApply` field was set to 1. To do so, it first filters the slices' data structures to obtain these slices. Next, it orders them by the number of lines that will be inserted from fixes (e.g., 1 for `scanf`, and 2 for `strncpy`), and for each resulting set, it sorts the slices by the line number of the sink in the original program, indicated in their data structure. For example, based on Fig. 3 it would obtain the sequence `S3:12` and `S2:9,10`. Afterwards, it deploys the correction and produces a new and fixed release of the PUT. Fig. 1 (b), at its bottom, shows the initial and vulnerable PUT fixed.

V. CorCA Implementation

The `CorCA` tool [1] was implemented in Python and has five main modules – *vulnerability finder*, *executable generator*, *vulnerability & fix validator*, *code corrector*, and *program release generator* –, where each one can work independently of the others. It integrates the Flawfinder [64] and AFL [65] tools to facilitate the realization of some specific steps of the tool pipeline, and resorts of the Pycparser [66] parser.

Flawfinder is a static analysis tool that scans C/C++ source code and reports potential security flaws. It is used by the vulnerability finder in the sinks finder step to signalize sinks associated with BOs. AFL is a fuzzer for C/C++ programs to exploit vulnerabilities they have, producing the test cases that exploit them by mutating the initial input and the ones produced from it. It is used in the vulnerability & fix validator module to confirm the existence of the potential vulnerabilities (vulnerability checker step) and to validate the fixes (fix validator step) generated by our approach. Pycparser [66] is a parser for the C language, written in Python, that parses the C code into an AST. It is used in different modules where is necessary to parse the code under analysis, namely in the vulnerability finder (slices extractor step), executable generator (slice program producer), vulnerability corrector (all steps), and program release generator (identify the right places step).

VI. Experimental Evaluation

The objective of this section is to evaluate the tool, but first, it is necessary to reason about the challenges stated in Section III and the solutions we proposed to solve them. Based on that information, we should evaluate four main abilities of the tool: find vulnerabilities, build compilable and executable files, correct vulnerabilities, and the effectiveness of the generated fixes. Considering these aspects, we defined the following questions: Q1. Is `CorCA` capable of detecting potential vulnerabilities associated with buffer overflows? Q2. Can `CorCA` extract correct code slices for the potential vulnerabilities found? Q3. Is the tool capable of generating compilable and executable files for the slices created? Q4. Can `CorCA` distinguish which potential vulnerabilities are real? Q5. Is the tool capable of correcting the vulnerabilities? Q6. Are the fixes generated by `CorCA` effective? Q7. Is `CorCA` capable of processing real applications and fixing vulnerabilities?

A. Evaluation Setup and Metrics

To evaluate the `CorCA`'s capabilities more thoroughly, we divided the evaluation into two parts. Firstly, we used a synthetic dataset of small C programs, taken from SARD [67], to evaluate the tool's performance and validate its abilities.

All programs of the dataset were previously classified manually as vulnerable ($Vuln$) or not vulnerable ($NotVuln$), serving as the ground truth for comparison with the results obtained by the tool. With this data, we created the confusion matrix, as the one presented in Table II, to calculate the next evaluation metrics to assess the tool's performance.

- *Accuracy, acc*: measures the percentage of correct decisions made by the tool. $acc = (TP + TN)/(P + N)$
- *False negative rate, fnr*: gets the percentage of vulnerable cases missed by the tool. $fnr = FN/(FN + TP)$
- *false positive rate, fpr*: measures the percentage of not vulnerable programs incorrectly identified as vulnerable by the tool. $fpr = FP/(FP + TN)$
- *Precision, pr*: gets the percentage of vulnerable programs correctly identified by the tool. $pr = TP/(TP + FP)$
- *Recall, rec*: measures the percentage of vulnerable cases the tool identified as such. $rec = TP/(TP + FN)$
- *Specificity, spc*: gets the percentage of not vulnerable cases that the tool identified. $spc = TN/(TN + FP)$
- *F-Score*: it is the harmonic mean of precision and recall. *F-Score* $= 2 * (pr * rec)/(pr + rec)$

Finally, in the second phase, we used real applications written in C taken from the SourceForge repository and from a project partner to test the tool's capabilities to process real programs. At this stage, the metrics presented above were not calculated because there is no classification of the applications to compare with the results obtained by the tool.

B. Evaluation with SARD dataset

SARD is a dataset that contains several small programs for types of vulnerabilities in C programs. From it, we gathered 1075 cases regarding with BOs and having each one 100 lines of code (LoC). The cases contain the functions `CorCA` addresses and were manually classified as vulnerable or not

TABLE II: General confusion matrix.

		Tool classification	
		Vuln	**Not Vuln**
Ground Truth	**Vuln**	TP	FN
	Not Vuln	FP	TN
	Total	P	N

TP: True Positives; TN: True Negatives; FP: False Positives
FN: False Negatives; P: Total Positives; N: Total Negatives

TABLE III: Summary of test cases collected from SARD.

	Function	Cases		Function	Cases
Input	gets	33	**Output**	sprintf	56
	scanf	120		vsprintf	30
	sscanf	120	**Data manipulation**	strcpy	115
	fscanf	120		strcat	115
	vscanf	30	**Multiple Functions**		276
	vsscanf	30			
	vfscanf	30	**Total**		1075

vulnerable to build the ground truth dataset. In total, we have 560 vulnerable cases and 515 not vulnerable cases. Table III summarizes the dataset by each function type.

The tool processed all instances and generated all slices and their executable correctly. Based on the knowledge from the ground truth dataset and the results obtained by the `CorCA` tool, Table IV was populated and represents the confusion matrix of the Vulnerability Finder, Vulnerability & Fix Validator (VF Validator), and Code Corrector modules. From this table, we calculated for each module the metrics defined in Section VI-A. Table V shows these metrics, from which it is possible to visualize the evolution throughout the tool's pipeline. Note that the VF Validator values shown in the table are relative to the first execution of this module in the tool's pipeline, i.e., the confirmation of vulnerability existence.

Based on results in Table IV, we verified, as expected, that the Vulnerability Finder module identified all programs as vulnerable. Since all programs have at least one sink we want to address, then this shows the module can identify these sinks. Furthermore, by analyzing the recall value of this module stated in Table V, we can observe that all the vulnerable programs were identified as such. Therefore, through these two verifications, we can infer that the tool is capable of finding potential BO vulnerabilities, answering affirmatively to Q1.

Regarding the Executable Generator module, we conclude that it fulfilled its role, i.e., it was able to produce compilable and executable slice programs correctly. An indicator of this result is that all `CorCA` modules were able to process all programs. In addition, we observed that VF Validator detected and exploited correctly all 560 vulnerable cases, which indicates that the executable files were correctly built and contained the slices with vulnerabilities. Furthermore, the module can invalidate the FP produced by the Vulnerability Finder, which is something we want this module to do. Hence, even though the finder has the highest fpr, the validator achieves both fnr and fpr null, i.e., a precision, recall, and F-Score equal to 100%. These results allow us to give a positive answer to questions Q2 and Q3 since these two are related.

TABLE IV: Confusion matrix of the modules evaluated.

		`CorCA` Classification					
		Vulnerability Finder		VF Validator		Code Corrector	
		Vuln	Not Vuln	Vuln	Not Vuln	Vuln	Not Vuln
Ground Truth	**Vuln**	560	0	560	0	560	0
	Not Vuln	515	0	0	515	30	485
	Total	1075	0	560	515	590	485

TABLE V: Summary of the calculated evaluation metrics.

Metric	Vulnerability Finder	Vulnerability Validator	Code Corrector
Accuracy (acc)	0.52	1.00	0.97
False Negative Rate (fnr)	0.00	0.00	0.00
False Positive Rate (fpr)	1.00	0.00	0.06
Precision (pr)	0.52	1.00	0.95
Recall (rec)	1.00	1.00	1.00
Specificity (spc)	0.00	1.00	0.94
F-Score	0.69	1.00	0.97

```c
1   #include <stdio.h>
2   #include <string.h>
3   #include <stdlib.h>
4   #define MAXSIZE 40
5
6   int main(int argc, char **argv) {
7     char userstr[MAXSIZE];
8     fgets(userstr, MAXSIZE, stdin);
9     int size = strlen(userstr) + 1;
10    char *userstr_copy = malloc(sizeof(char) * size);
11    strcpy(userstr_copy, userstr);
12    puts(userstr_copy);
13    free(userstr_copy);
14    return(0);
15  }
```

Listing 2: Example of the first reason for false positives.

The results obtained by the VF Validator depicted in both tables show that this module was able to correctly identify all vulnerable and not vulnerable cases, which serve to answer Q4 since this module detected all the real vulnerabilities.

As mentioned before, the results showed to VF Validator are relative to its first execution in the tool's pipeline. In its second execution (the fix validator), all cases were classified as not vulnerable since the module did not find any problems during the fix validation. This result indicates that the Code Corrector module was able to generate correct syntactically and effective corrections as they removed the existing vulnerabilities. Therefore, we can answer questions Q5 and Q6, meaning that the `CorCA` was able to generate the right corrections and insert them in the right places into the code. After all, the generated corrections proved to be effective.

From the results of both tables, we also verified that the Code Corrector had 30 FP, i.e., it corrected some programs that are not vulnerable, that are due to two major reasons: (i) the limitation of static analysis to deal with values calculated at runtime, and (ii) a string is truncated before being used in data manipulation sensitive sinks (e.g., `strcpy` or `strcat`). In both situations, a correction would always be performed, for prevention, to avoid possible FNs. Overall, the Code Corrector had a recall of 100 % and a precision of 95 %. An example of the first situation is depicted in Listing 2, which is observed in some SARD cases that contain allocation of dynamic buffers, whose size is calculated based on the result returned by executing some function. In that cases, it is not possible to statically determine the size of the buffer. The function `fgets` reads some text provided by the user and writes it into the buffer `userstr` (line 8). Next, the length of this text is determined and stored in `size` and added a byte to it (line 9). Using this value, the program allocates memory to the char pointer `userstr_copy`, to which the content of the buffer `userstr` is copied (lines 10 and 11). There is no buffer overflow because the `userstr_copy` was created with enough size to store the contents of `userstr`. However, through static analysis, it is not possible to determine the `size` value because it is calculated at runtime. Therefore, it is not possible to compare the `strcpy` function parameters to check whether there is a buffer overflow, which leads the Code Corrector module to fix this code.

C. Evaluation with Real Applications

SARD dataset allowed the evaluation of the `CorCA` capability to deal with all the functions addressed and to measure its performance. However, SARD's cases might not represent real applications accurately. Therefore, to test our tool with real code, we obtained 6 applications from SourceForge of different contexts (e.g., network, MAC generator) and a railway driver software from a propulsion control system (PCS) from a partner of the project in which this work is inserted. Table VI presents a summary of the applications we tested (columns 2 and 3), namely their number of files and LoC, and the results of the evaluation (last two columns). In total, the tool analyzed 209 files, corresponding to 132,209 LoC, flagged 59 potential vulnerabilities and fixed 6, which correspond to zero-day vulnerabilities. The symbol (-) in the table means that the tool could not finish its entire process. Note that some of these applications are early-stage development versions and sometimes present some problems or even are incomplete. In this case, it was necessary to modify some files manually because some include libraries were incorrect.

For the *Zervit* and *Tiny HTTPd* applications, and the PCS's driver, the tool signalized 6, 38 and 4 potential vulnerabilities, respectively. For the 6, it generated 4 fixes, for the 38, it generated 2 fixes, and for the 4, it generated no fixes. These 6 fixes were manually verified, and we concluded that they were necessary and correctly removed the vulnerabilities found. This result shows that it was possible to discover 6 vulnerabilities that were not yet reported for these versions of the applications, i.e., the tool discovered 6 zero-day vulnerabilities. The remaining 42 potential vulnerabilities found were in fact not vulnerable, which the Vulnerability & Fix Validator confirmed, thus invalidating the 42 FPs provided by the Vulnerability Finder. For the *sSocks* application, the tool was able to detect 10 potential vulnerabilities but was unable to proceed with the process because some include libraries were not provided with the program, making the parser incapable of parsing some files. The same problem occurred with *Intel Ethernet Drivers*, although the tool had detected 1 potential vulnerability. In the case of the *LIBPNG* and *Macgen* applications, the tool did not find any potential vulnerabilities associated with the addressed sensitive sinks. Therefore, it did not generate any fixes.

TABLE VI: Real applications evaluation of `CorCA`.

Application	Files	LoC	Potential Vulnerabilities	Fixes Generated
Zervit 0.4	17	1014	6	4
Macgen 1.1	1	15	0	0
sSocks 0.0.14	30	3477	10	(-)
Tiny HTTPd 0.1	3	765	38	2
LIBPNG 1.6.37	88	57075	0	0
Intel Ether Drs 2.17.4	50	64380	1	(-)
PCS's Driver	20	5483	4	0

Given these results, we can positively answer Q7, meaning that `CorCA` is capable of processing real applications and fixing vulnerabilities in them.

VII. Threats to Validity

Following Cook and Campbell [68] we assessed the results and conclusion for the threats to validity. The process conducted in the experiments to validate and evaluate `CorCA` was, respectively, based on a set of small C programs containing BOs and some real applications that we did not have any knowledge of their security. We desired to find out the ability of the tool to find vulnerabilities, even previously unknown vulnerabilities (zero-days), process more complex programs, and correct the code effectively. In the validation phase, the tool achieved a high level of confidence, with accuracy, precision, and an F-score equal or close to 1. Such metrics reside in a ground truth dataset of 1075 SARD programs that we manually characterized and labeled as vulnerable and non-vulnerable. Based on these metrics, we concluded that the tool has achieved the purpose for which it was designed. For the evaluation of real applications, the tool presented some limitations in parsing the PUT's code, some due to missing PUT files and others related to its parsing process, denoting thus that it needs improvements. However, the tool was able to detect and fix vulnerabilities correctly on real programs.

VIII. Conclusions

The paper explores a new form to protect C programs from buffer overflows (BO) vulnerabilities. We proposed a fully automated solution to detect BOs, confirm their existence, fix these, and verify that the fixes generated are effective. It presents the idea to find candidate vulnerabilities through static analysis, then gather the code instructions associated with them and create executable slice programs containing the data flow path of each potential vulnerability. Later on, these programs are fuzzed to determine which vulnerabilities are real and that will be submitted to the process of code repairing and fix verification. We developed the `CorCA` tool to the proposed approach, and we validated it with a dataset from SARD and evaluated it with real applications, where it found 6 zero-day vulnerabilities. The experimental results showed that `CorCA` was able to detect BOs vulnerabilities and correct them effectively. Furthermore, it is beneficial in having a pipeline for finding BOs, confirming and correcting them, and testing the new code, which can be helpful for developers and improve the code quality and software security.

Acknowledgments

This work was partially supported by the national funds through P2020 with reference to the XIVT project (LISBOA-01-0247, FEDER-039238, an ITEA3 European project (I3C4-17039)), and through FCT with reference to LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

REFERENCES

[1] "CorCA tool and materials," https://github.com/iberiam/CorCA/.

[2] Tiobe Index, https://www.tiobe.com/tiobe-index/.

[3] 2021 CWE Top 25 Most Dangerous Software Weaknesses, https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[4] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, Aug. 2019, pp. 1967–1983.

[5] P. Oehlert, "Violating assumptions with fuzzing," in *IEEE Security and Privacy*, vol. 3, Number 2, no. 2, March 2005, pp. 58–62.

[6] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling Black-box Mutational Fuzzing," in *Proceedings of the ACM SIGSAC Conference on Computer, Communications Security*, Nov 2013, pp. 511–522.

[7] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, pp. 20–27, jan 2012.

[8] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2013, pp. 122–131.

[9] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct 2018, pp. 2095–2108.

[10] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the USENIX Security Symposium (USENIX Security 13)*, Aug 2013, pp. 49–64.

[11] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems*, Dec. 2019, pp. 10 197–10 207.

[12] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Annual Network and Distributed System Security Symposium*, Feb. 2018.

[13] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, 2007, p. 329?340.

[14] A. Smirnov and T.-c. Chiueh, "Automatic Patch Generation for Buffer Overflow Attacks," in *Proceedings of the 3rd International Symposium on Information Assurance and Security*, 2007, pp. 165–170.

[15] F. Gao, L. Wang, and X. Li, "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, 2016.

[16] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications," *ACM SIGPLAN Notices*, vol. 50, no. 6, p. 43?54, 2015.

[17] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[18] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, no. 1, jan 2018.

[19] "NIST Software Assurance Reference Dataset Project," https://samate.nist.gov/SARD/.

[20] "Internet Security Glossary," https://www.ietf.org/rfc/rfc4949.txt.

[21] "ISO/IEC 27000:2018," https://www.iso.org/standard/73906.html.

[22] "Vulnerability - Glossary - CSRC," https://csrc.nist.gov/glossary/term/vulnerability.

[23] "CVE-1999-0002 : Buffer overflow in NFS mountd," https://www.cvedetails.com/cve/CVE-1999-0002/.

[24] "CVE-2004-0234 : Multiple stack-based buffer overflows," https://www.cvedetails.com/cve/CVE-2004-0234/.

[25] "CVE-2011-1270 : Buffer overflow in Microsoft PowerPoint 2002 SP3," https://www.cvedetails.com/cve/CVE-2011-1270/.

[26] "CVE-2020-25583: In FreeBSD 12.2-STABLE," https://www.cvedetails.com/cve/CVE-2020-25583/.

[27] Y. Younan, W. Joosen, and F. Piessens, "Code injection in C and C++ : A survey of vulnerabilities and countermeasures," Departement Computerwetenschappen, Katholieke Universiteit Leuven, Tech. Rep., 2004.

[28] K. Piromsopa and R. J. Enbody, "Survey of Protections from Buffer-Overflow Attacks," *Engineering Journal*, vol. 15, pp. 31–52, 2011.

[29] E. H. Boudjema, C. Faure, M. Sassolas, and L. Mokdad, "Detection of security vulnerabilities in C language applications," *Security and Privacy*, 2017.

[30] J. Kronjee, "Discovering Software Vulnerabilities Using Data-Flow Analysis and Machine Learning," Master's thesis, Open University, faculty of Management, Science and Technology, 2018.

[31] L. Flynn, W. Snavely, D. Svoboda, N. VanHoudnos, R. Q. J. Burns, D. Zubrow, R. Stoddard, and G. Marce-Santurio, "Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models," *International Workshop on Software Qualities and their Dependencies*, 2018.

[32] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning," *In Proceedings of the USENIX Workshop on Offensive Technologies*, 2011.

[33] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *In Proceedings of the Symposium on Network and Distributed Systems Security*, 2003.

[34] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[35] K. Vorobyov, N. Kosmatov, and J. Signoles, "Detection of Security Vulnerabilities in C Code using Runtime Verification: an Experience Report," *In Proceedings of the International Conference on Tests And Proofs*, 2018.

[36] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, p. 58?62, 2005.

[37] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 2008.

[38] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling Black-Box Mutational Fuzzing," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, 2013, p. 511?522.

[39] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," in *Proceedings of the International Conference on Software Engineering*. IEEE Press, 2013, p. 122?131.

[40] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, p. 20?27, 2012.

[41] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-Box Fuzzer," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2018, p. 2095?2108.

[42] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers," in *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019, p. 1967?1983.

[43] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Conference on Security*. USENIX Association, 2013, p. 49?64.

[44] "LibFuzzer," https://llvm.org/docs/LibFuzzer.html.

[45] "Honggfuzz," https://github.com/google/honggfuzz/.

[46] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 91–101.

[47] M. Fang and M. Hafiz, "Discovering buffer overflow vulnerabilities in the wild: An empirical study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014.

[48] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," *arXiv*, 2018.

[49] G. Griecox, L. Grinblatx, L. Uzalx, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using Machine Learning," *In Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, 2016.

[50] Z. Li, D. Zou, S. Xux, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability De-

tection," *In Proceedings of the Network and Distributed Systems Security Symposium.*, 2018.

[51] W. A. Dahl, L. Erdodi, and F. M. Zennaro, "Stack-based Buffer Overflow Detection using Recurrent Neural Networks," *arXiv*, 2020.

[52] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019.

[53] E. Pinconschi, R. Abreu, and P. Adão, "A comparative study of automatic program repair techniques for security vulnerabilities," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 196–207.

[54] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering*, 2016.

[55] S. Ding, H. B. K. Tan, and H. Zhang, "Automatic removal of buffer overflow vulnerabilities in C/C++ programs," *In Proceedings of the 16th International Conference on Enterprise Information Systems*, vol. 2, pp. 49–59, 2014.

[56] R. Morgado, I. Medeiros, and N. Neves, "Towards web application security by automated code correction," in *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2020.

[57] H. Shahriar, H. M. Haddad, and I. Vaidya, "Buffer Overflow Patching for C and C++ Programs: Rule-Based Approach," *ACM SIGAPP Applied Computing Review*, vol. 13, no. 2, p. 8?19, 2013.

[58] Z. Chen, S. Kommrusch, and M. Monperrus, "Using Sequence-to-Sequence Learning for Repairing C Vulnerabilities," *arXiv*, 2019.

[59] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *In Proceedings of the ACM on Programming Languages*, 2019.

[60] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural Program Repair by Jointly Learning to Localize and Repair," *In Proceedings of the International Conference on Learning Representations*, 2019.

[61] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon, "Learning to Catch Security Patches," *arXiv*, 2020.

[62] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, p. 101?114.

[63] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, 2021.

[64] "Flawfinder," https://github.com/david-a-wheeler/flawfinder.

[65] "American Fuzzy Lop (AFL)," https://github.com/google/AFL.

[66] "Pycparser," https://github.com/eliben/pycparser.

[67] "NIST Software Assurance Reference Dataset Project," https://samate.nist.gov/SARD/.

[68] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.