IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 18, 2021, accepted October 5, 2021, date of publication October 15, 2021, date of current version October 26, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3120349

Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects

JOSÉ D'ABRUZZO PEREIRA, (Member, IEEE), NAGHMEH IVAKI, (Member, IEEE), AND MARCO VIEIRA, (Member, IEEE)

Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, University of Coimbra, 3030-790 Coimbra, Portugal

Corresponding author: José D'Abruzzo Pereira (josep@dei.uc.pt)

This work was supported by the Portuguese Foundation for Science and Technology (FCT) under Grant 2020.04503.BD, in part by the Project METRICS through FCT under Grant POCI-01-0145-FEDER-032504, in part by the Project "AIDA—Adaptive, Intelligent and Distributed Assurance Platform" co-financed by the European Regional Development Fund (ERDF) and COMPETE 2020 under Grant POCI-01-0247-FEDER-045907, and in part by FCT under Carnegie Mellon University (CMU) Portugal.

ABSTRACT Security vulnerabilities are present in most software systems, especially in projects with a large codebase, with several versions over the years, developed by many developers. Issues with memory management, in particular buffer overflow, are among the most frequently exploited vulnerabilities in software systems developed in C/C++. Nevertheless, most buffer overflow vulnerabilities are not detectable by vulnerability detection tools and static analysis tools (SATs). To improve vulnerability detection, we need to better understand the characteristics of such vulnerabilities and their root causes. In this study, we analyze 159 vulnerable code units from three representative projects (i.e., Linux Kernel, Mozilla, and Xen). First, the vulnerable code is characterized using the Orthogonal Defect Classification (ODC), showing that most buffer overflow vulnerabilities are related to *missing* or *incorrect checking* (e.g., missing *if* construct around statement or incorrect logical expression used as branch condition). Then, we run two widely used C/C++ Static Analysis Tools (SATs) (i.e., CppCheck and Flawfinder) on the vulnerable and neutral (after the vulnerability fix) versions of each code unit, showing the low effectiveness of this type of tool in detecting buffer overflow vulnerabilities. Finally, we characterize the vulnerable and neutral versions of each code unit using software metrics, demonstrating that, although such metrics are frequently used as indicators of software quality, there is no clear correlation between them and the existence of buffer overflow in the code. As a result, we highlight a set of observations that should be considered to improve the detection of buffer overflow vulnerabilities.

INDEX TERMS Software security, buffer overflow, static code analysis, vulnerability detection, orthogonal defect classification (ODC), software metrics.

I. INTRODUCTION

Most computing systems suffer from software vulnerabilities, a particular type of defect that may open the door to security attacks [1]. When an attack happens, severe damages may occur, such as gain of administrative privileges, access to confidential information, financial loss, and even safety-related violations.

Since 2020 we have witnessed a steep increase in the relevance of software security due to the COVID-19 pandemic, as most businesses and organizations have to be available over the Internet to support online working and services more than ever [2]. This has created more possibilities for security attacks, as systems that were developed without considering strict security requirements can no longer be restricted to corporate networks.

According to *Steve Zurier*, most IT leaders intend to spend more than 40% of their 2021 budget in cybersecurity to avoid potential attacks [3]. Moreover, a White Source report states that C and C++ account for 52% of vulnerabilities in open source software (C = 46%; C++ = 6%) [4]. Among these vulnerabilities, **improper use of memory**, which may lead to **buffer overflows**, is the most frequent type of vulnerability in C/C++ code [4].

The secure coding practices from the Open Web Application Security Project (OWASP) [5] and the coding standards from the Software Engineering Institute (SEI) CERT [6], among many other resources available, provide fundamental

The associate editor coordinating the review of this manuscript and approving it for publication was Neetesh Saxena.

guidelines and checklists to handle memory management properly. Despite that, many software systems are still released with vulnerabilities of this type, which are usually caused by the lack of knowledge and experience on applying coding practices and guides throughout the Software Development Lifecycle (SDLC).

Barry Boehm highlights, in his famous ''Software Engineering Economics'' book, that ''the earlier the defects are discovered, the cheaper it is to fix them'' [7]. Security vulnerabilities, in particular memory management-related issues, are a type of software defect for which Boehm's statement is particularly valid. In fact, from a business perspective, the consequences of vulnerabilities being exploited can be even more severe than of ''classical'' software defects. This way, effective techniques and tools to detect software vulnerabilities must be applied from the early stages of software development.

Common and widely-used vulnerability detection techniques are based on Static Code Analysis [8] and on the analysis of Software Metrics (SMs) [9]. Static Analysis Tools (SATs) analyze the code (without executing it) and report alerts indicating potential issues, including security vulnerabilities. The software development team should then verify these alerts and fix the correctly detected issues. On the other hand, SMs can be analyzed using techniques such as Machine Learning (ML) [9]–[11] and genetic algorithms [12] to identify potentially vulnerable code units (e.g., files, functions, and classes that probably are vulnerable). In this case, the software development team should review, analyze (e.g., using SATs), and/or test the indicated code units to find and fix the possible vulnerabilities.

Both SATs and SMs have limitations in what regards the detection of software vulnerabilities. On one hand, although considered effective to point the problematic (or vulnerable) lines of code, SATs are known for reporting a high number of false alarms [13]. On the other hand, SMs combined with ML may achieve good results in terms of avoiding false alarms, but creating precise and generic (i.e., not overfitted) prediction models is quite challenging [14]. Also, these techniques are usually weaker for buffer overflow vulnerabilities [15]. To improve the current situation, we need to better understand how buffer overflow vulnerabilities are fixed, and what are the capabilities and limitations of using SATs and SMs to detect such weaknesses. SATs indicates potential problems in the source code, while SMs reveal code structural characteristics. This helps to research new approaches for vulnerability detection or improve the existing ones (e.g., by creating new detection rules for SATs).

This work studies the **characteristics of code with buffer overflow vulnerabilities and of the fixes applied to remove them**. To support this study, we rely on SMs and SAT alerts as they are frequently used to collect information about the quality of the code during the development phase [16]. Moreover, they can be easily extracted from the code under development. Although this type of study is also important for other types of vulnerabilities, they should be treated independently, as different conclusions may be reached depending on the specific characteristics of each type.

We analyze the vulnerable version and the neutral version (after a code fix) of a set of 159 code units from three representative C/C++ open-source projects (i.e., Linux Kernel, Mozilla, and Xen) in an attempt to cast light on the following Research Questions (RQs):

- RQ1: What are the main changes in the code when fixing buffer overflow vulnerabilities?
- RQ2: What are the differences between SAT alerts reported before and after fixes?
- RQ3: What is the impact of buffer overflow vulnerability fixes on SMs that portray code characteristics?

The analysis follows three complementary directions: *i)* **study the changes in the source code when fixing buffer overflow vulnerabilities** by applying the Orthogonal Defect Classification (ODC); *ii)* **study the SAT alerts in the vulnerable and neutral versions of code units** by running two widely known C/C++ SATs (CppCheck and Flawfinder); and *iii)* **understand the eventual correlation of SMs with the existence of buffer overflow vulnerabilities** by comparing their variation between the vulnerable and neutral versions of the code. The outcome is a set of key observations that can be used in the future to improve the detection of buffer overflow vulnerabilities.

The rest of this paper is organized as follows. Section II presents background concepts and related work. Section III describes the approach and the experiments conducted. The analysis of the main code changes due to vulnerability fixes and the ODC classification are presented in Section IV. Section V presents the analysis of the SAT alerts and Section VI focuses on the impact of the fixes on SMs. Section VII discusses the limitations of the work and highlights the threats to validity of the results. Finally, Section VIII concludes the paper and presents directions for future work.

II. BACKGROUND AND RELATED WORK

This section presents concepts on software vulnerability (in particular, buffer overflow), techniques for vulnerability detection, and related works.

A. SOFTWARE VULNERABILITY AND BUFFER OVERFLOW

According to *ISO/IEC 27000:2018* [1], a vulnerability is a ''weakness of an asset or control that can be exploited by one or more threats''. Such weakness, which can be a design flaw or an implementation bug, leaves the software system vulnerable to attacks. A successful attack may lead to safety violations (i.e., harmful to the system, environment, or human life), data breaches, financial loss, among other consequences [17], depending on the type of vulnerability exploited and the intentions of the attackers.

Buffer overflow is a weakness related to inadequate memory management. It occurs when the software allows reading or writing in a memory space outside the allocated

memory [18]. A typical example of a buffer overflow weakness is the use of the C function `strcpy`, as it copies a string from a memory location to another without checking the boundaries of the destination buffer. As an example, one of the vulnerable code snippets of the Linux Kernel project analyzed in this study is shown below (`sound/oss/soundcard.c` before the vulnerability fix CVE-2010-4527). It uses the `strcpy` function in an old version of the Linux Kernel. This vulnerability can be fixed either by replacing the vulnerable function (i.e., `strcpy`) by another function, such as `strlcpy`, or by adding a checking before the use of `strcpy`.

```
n = num_mixer_volumes++;

strcpy(mixer_vols[n].name, name);

if (present)
  mixer_vols[n].num = n;
else
  mixer_vols[n].num = -1;
```

The **Common Weakness Enumeration (CWE)** [19] provides a community-developed list of known software and hardware weaknesses/vulnerabilities for a large set of systems. In this categorization system, most memory management-related vulnerabilities are classified under the *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer* identifier [18]. Although this is a known problem, it is still the most frequent weakness found in most C/C++ software systems for several reasons: *i)* there is no built-in protection in C/C++ applications against accessing/overwriting data in memory; *ii)* C/C++ software systems do not automatically check whether data written to a buffer respects the bounds of that buffer; and *iii)* several and different possibilities and paths in the code may lead to the buffer overflow issue, which makes it challenging to find all possibilities, especially in complex and large projects.

B. VULNERABILITY DETECTION TECHNIQUES

There are several techniques to discover software vulnerabilities, which can be divided into dynamic and static techniques [20]. Dynamic techniques, such as Software Penetration Testing (SPT) [21], involve the execution of the software system and test it against simulated (or emulated) attacks. When an attack is well-succeeded, a vulnerability is detected and exploited. On the other hand, static techniques, which are the main focus of this study, try to find the potential vulnerabilities by analyzing the source code without its execution [22].

Although SATs are not limited to the identification of security vulnerabilities, one of their main uses is for this purpose due to the severe consequences that vulnerabilities may have. The static analysis is normally performed by following and checking a set of rules specified for a specific programming language. Examples of SATs are Spotbugs [23] for Java (formerly known as Findbugs), Microsoft FxCop for .Net [24], Pixy [25] for PHP, and Parasoft CppTest [26] for C/C++. Some SATs include rules for several programming languages, such as SonarQube [27] (27 programming languages, such as Java, C#, C/C++, Python, and PHP) and Coverity [28] (21 programming languages, such as C/C++, Java, Scala, and PHP).

The techniques to identify the issues are diverse, including syntactic pattern matching, lexical analysis, data flow analysis (and its particular case of taint analysis), parsing, model checking, and symbolic execution [29], [30]. Each issue identified and reported by a SAT is called an "alert". An alert includes the following main attributes: *i)* a type (source of the problem), *ii)* a filename, and *iii)* the line of code in which the alert is raised. Other attributes can also be reported, such as the alert severity, category of the vulnerability, Common Weakness Enumeration (CWE), and a message or a description. In practice, each SAT defines some additional attributes to report.

SMs can also be used to detect software vulnerabilities. This is usually achieved by analyzing the architectural measures of the source code through different types of metrics, such as *i)* volume (e.g., Lines of Codes (LOCs)), *ii)* coupling (e.g., coupling between objects/classes), *iii)* cohesion (e.g., lack of cohesion), and *iv)* complexity (e.g., McCabe's cyclomatic complexity [31]). SMs can be analyzed using ML algorithms [9], [11] or genetic algorithms [12] to indicate low-quality code units (i.e., files, functions, classes) that may be (with some probability) vulnerable. The software development team is then responsible for reviewing the code to identify the exact location of possible vulnerabilities.

C. RELATED WORK

Arusoaie *et al.* [32] benchmark 11 distinct open source C/C++ SATs using 638 test cases from a test suite created by Toyota [33]. Their results show that Clang has the highest detection rate, with a value of 0.358. Although the test suite includes an extensive number of test cases, they are not from real projects. Their goal is to benchmark the SATs using a well-defined set of vulnerabilities that can potentially be detected by the tools, and the work does not analyze the vulnerabilities that could not be detected. Nevertheless, results clearly show that SATs have great limitations.

Nong et al. evaluate vulnerability detection tools (both static and dynamic analysis tools) to identify memory-related vulnerabilities [34]. They also use the dataset created by Toyota, with the same 638 test cases and consider one SAT (i.e., CBMC) and four dynamic analysis tools (i.e., AddressSanitizer, Valgrind, MemorySanitizer, DrMemory). Their results show that SATs accuracy needs to be improved and that it is difficult to obtain both good precision and recall for the same tool. Moreover, tools that use a hybrid approach (static and dynamic techniques) usually have better accuracy. Once again, due to the limitations of the dataset selected, it may not be representative of vulnerabilities in real projects.

Medeiros et al. use SMs to predict vulnerable code with ML algorithms (Decision Tree (DT), Random Forest (RF), Extreme Gradient Boosting (XGB), and Support Vector

Machine (SVM)) at two levels of code units (file and function) [14] of five C/C++ projects (Mozilla, Linux Kernel, Xen, httpd, and glibc [11], [35]). The algorithms with the best performance at the file level are RF and XGB, in all the scenarios evaluated. The file-level results are usually better (recall about 0.900) than the function-level (recall about 0.800). Nevertheless, the prediction does not take into consideration the vulnerability type. Additionally, the work suffers from the problem of the use of SMs: the prediction indicates potentially vulnerable code without indicating the exact place of the vulnerability. Furthermore, the good results for recall hide a low precision, meaning that a large number of false alarms are raised. Consequently, additional time needs to be spent by development teams to identify the true vulnerabilities.

Walden et al. use static data as input (features) for several ML algorithms (i.e., DTs, k-Nearest Neighbors (k-NN), Naive Bayes (NB), RFs, and SVM) to detect software vulnerabilities [9]. Two types of features (SMs and text mining) are extracted from PHP applications (Drupal, Moodle, and PHPMyAdmin). Overall, their results are better when using the text mining features in the three projects (recall varying from 0.737 to 0.805, while with SMs, it ranges from 0.663 to 0.769). This is probably related to the small number of SMs considered, which is 12. The results may also be overfitted for text mining, which leads to better performance.

Morrison et al. created ODC+V, which is an extension of ODC tailored to classify software vulnerabilities [36]. As the standard ODC has a single value, ''security/integrity'', for the ''impact'' attribute, authors claim that it is not possible to characterize the impact of vulnerabilities in a precise way. ODC+V proposes a new attribute called ''security impact'', which may have one of the following values from Microsoft STRIDE [37]: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. The work presents an evaluation considering 583 defects and 583 vulnerabilities from three projects (Firefox, PHPMyAdmin, and Chrome). The goal is to compare if defects and vulnerabilities are fixed and discovered in the same manner. Results show that vulnerabilities are usually found later in the SDLC compared to ''classical'' defects. Moreover, vulnerabilities are primarily classified in one of the following ODC defect types: Checking, Assignment/Initialization, or Algorithm/Method. Furthermore, vulnerabilities are often fixed by adding a checking condition (such as an if-clause) than other defects. The use of STRIDE in the ''impact'' attribute has a side-effect: ODC+V is not orthogonal (like the standard ODC) as each attribute may have more than one value. As we are interested in understanding the root causes and fixes of buffer overflow vulnerabilities and not their impact, the standard ODC is adequate for the present work.

Zheng et al. evaluate the capability of static techniques to detect faults [38]. Three Nortel projects written in C/C++ are analyzed using the data from three SATs (FlexeLint, Klockwork, and Reasoning's Illuma) previously included in the Nortel inspection process. They use ODC to identify the faults and failure types detected by the three techniques studied (static analysis, inspection, and testing). The results indicate that testing is two to three times more effective than static code analysis and inspection, which have similar performance. Additionally, the used SATs are effective at identifying two ODC defect types, assignment and checking. Their results also indicate that SATs can be used to find vulnerabilities caused by programming errors. However, their findings are limited to the projects of a single organization (Nortel) and may not be valid in other contexts.

Li et al. created the VulDeePecker, a deep learning-based approach to detect vulnerabilities [39]. Code gadgets, which are a vector representation of the functions, are used as input for the deep learning algorithm. The dataset has 61, 638 code gadgets, 17, 725 of which are vulnerable: 10, 440 code gadgets with buffer errors (CWE-119), and 7, 285 with resource management errors (CWE-399). Their evaluation uses vulnerabilities of three C/C++ software projects (Xen, Seamonkey, and Libav), and VulDeePecker could detect 4 vulnerabilities not reported in the National Vulnerability Database (NVD) [40]. Nevertheless, these vulnerabilities were silently fixed in future versions of the evaluated software projects. Their results also show a lower false positive rate (FPR) (5.7%) and false negative rate (FNR) (7.0%), when compared to the other techniques evaluated in the study. A key limitation of VulDeePecker is that it only deals with vulnerabilities related to library/API function calls (e.g., `strcmp`).

Jia et al. propose an offline analysis solution called HOTrace to identify heap vulnerabilities [41]. To do that, HOTrace uses programs' execution traces and identifies taint attributes during the execution. The whole process is done in the programs themselves, without the source code. The evaluation was performed in 17 Windows x86/x64 applications. Using their prototype, they identified 47 previously unknown heap overflow vulnerabilities, including two vulnerabilities in Microsoft Word.

Haller et al. created the fuzzer Dowser to detect buffer overflow violations [42]. To do that, they combine taint tracking, program analysis, and symbolic execution. Dowser ranks the source code to perform the analysis, which is done based on the data-flow graph. It uses the intuition that complex control flows are more prone to buffer overflow vulnerabilities. They also reduce the symbolic input using dynamic taint analysis to improve the performance of Dowser. The evaluation was performed in six applications (nginx, ffmpeg, inspircd, libexif, poppler, and snort). They could identify two previously unknown buffer overflow vulnerabilities.

Liu et al. analyzed five open source C/C++ projects (Linux Kernel, FFmpeg, ImageMagick, OpenSSL, and php-src) and presented 12 findings, which were applied to find 10 zero-day vulnerabilities [43]. The authors wanted to understand if more vulnerabilities can be found close to identified vulnerabilities. Using the commits to fix the identified vulnerabilities, they built the call-graph for the vulnerable code snippets. They found that the vulnerabilities usually

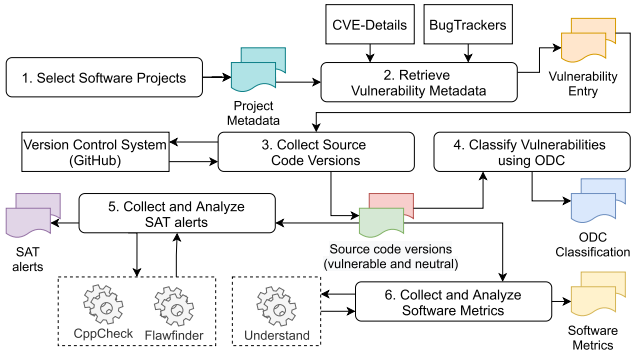

FIGURE 1. Approach for analysis of buffer overflow vulnerabilities.

follow the Pareto law and that 60% of the vulnerabilities have at least another one close to them (in a 2-jump range in the call-graph).

In general, the main limitation of the existing works is that they do not analyze a specific type of vulnerability, particularly buffer overflow, from the perspective of understanding how it is fixed and how effective static techniques are in its detection. To the best of our knowledge, no other study characterizes buffer overflow using the approach followed in this work: using ODC classification and the analysis of SAT alerts and SMs, in both vulnerable and neutral versions of code units.

III. APPROACH AND EXPERIMENTATION

The approach followed in our study, depicted in Figure 1, is composed of six steps: *i)* select several representative software projects, from a security perspective; *ii)* retrieve vulnerability metadata; *iii)* collect source code versions; *iv)* classify the vulnerabilities using ODC; *v)* collect and analyze SATs alerts; and *vi)* collect and analyze SMs. Each step is detailed in the following sections.

A. SELECT SOFTWARE PROJECTS

A dataset created and updated by Alves *et al.*, which includes the vulnerabilities reports for five open-source C/C++ projects between 2000 to 2016 [11], [35], is used as starting point for the analysis. The vulnerabilities were collected from CVE-Details,[1] and the dataset includes SMs for both vulnerable and neutral versions of each code unit (files, functions, and classes) of five projects. The SMs were calculated using the SciTools Understand [44] tool. From the five projects in the dataset, we selected the three with the largest codebase and with the highest number of known vulnerabilities:

1) **Linux Kernel**: open-source operating system kernel created by Linus Torvalds in 1991 (available in https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git);
2) **Mozilla**: codebase of the browser Mozilla Firefox, and other applications, e.g., the e-mail client Thunderbird (available in https://github.com/mozilla/gecko-dev);

[1]https://www.cvedetails.com

TABLE 1. Number of vulnerabilities: A) all vulnerabilities, B) vulnerabilities with the CWE-119 identifier, C) CWE-119 vulnerabilities with "overflow" in vulnerability type.

	Linux Kernel	Mozilla	Xen	Total
A	867 (100.0%)	2441 (100.00%)	148 (100.0%)	3456 (100.0%)
B	123 (14.2%)	336 (13.8%)	18 (12.2%)	477 (13.8%)
C	98 (11.3%)	56 (2.3%)	5 (3.4%)	159 (4.6%)

3) **Xen**: hypervisor that supports virtualization of environments with both Linux and Windows operating systems (available in https://xenbits.xen.org/gitweb/?p=xen.git).

The selected projects are representative from a security perspective as they are widely used and were frequently the target of security attacks during a long period of time, as it can be seen in CVE-Details, the three projects are listed as the top-50 vendors with distinct vulnerabilities.[2] The other two projects in the dataset (glibc and httpd) are not considered in this study, as they are small projects with a small number of reported vulnerabilities, thus, not allowing to draw relevant observations and conclusions.

CVE-Details provides several pieces of information about known vulnerabilities, including the Common Vulnerability Scoring System (CVSS), the impact, the vulnerability type (which can have multiple values), and the CWE (although not all vulnerabilities have a CWE assigned). As we are dealing particularly with "Buffer Overflow", we consider only the records with the identifier *CWE-119*, which represent between 12.2% to 14.2% of all vulnerabilities in each of the three projects. From the vulnerabilities with a *CWE-119* identifier, we only selected the ones whose type attribute includes the value "Overflow", which represent between 2.3% to 11.3% of the vulnerabilities in each project. This is shown in Table 1, which shows a quantitative summary of the vulnerabilities of the three selected projects (Linux Kernel, Mozilla, Xen).

Although the number of records for analysis is not large (only 159), it seems to be enough as we are dealing with a single type of vulnerability (CWE-119: Buffer Overflow). The vulnerabilities are from several versions of three C/C++ projects with different purposes and functionalities (an operating system, a browser, and a hypervisor) and were detected over a considerable period of time (from 2000 to 2016). These result in diverse samples with potentially different root causes, allowing to draw relevant observations and meaningful conclusions.

B. RETRIEVE VULNERABILITY METADATA

The information about the vulnerabilities was collected from CVE-Details. Each vulnerability entry has a unique identifier called CVE-ID, which is composed of the prefix "CVE", the year in which the vulnerability was reported, and a sequential number. Each CVE-ID includes basic information about the

[2]https://www.cvedetails.com/top-50-vendors.php

vulnerability, such as the CVSS, the impact, the vulnerability type (which can have multiple values), the CWE, and a table with all software versions affected by that vulnerability. For some vulnerabilities, information on how they can be exploited is also included.

Each CVE-ID was complemented with data from BugTrackers, which are software applications that are used to manage the issues reported for a project (e.g., Mozilla uses Bugzilla as BugTracker). BugTrackers include key information about the specific commit where a vulnerability was fixed.

C. COLLECT SOURCE CODE VERSIONS

The source code of the projects considered in this study is either stored in GitHub[3] or in a mirror of the repository available in GitHub. Knowing the commit that fixes a vulnerability (from the BugTracker entries), we obtained the neutral version of the source code (with the vulnerability fixed) and its previous version (the vulnerable one). With these, we can manually analyze the code and run the tools to extract the required information (e.g., SATs alerts and SMs).

D. CLASSIFY VULNERABILITIES USING ODC

The next step is to classify the vulnerabilities using ODC [45], which is a systematic approach to classify defects identified in a software system. It is widely used for root-cause analysis and defines several attributes to be filled when the defect is open and closed, based on a predefined set of values. The attributes registered when a defect is open include: i) activity, ii) trigger, and iii) impact. When the defect is closed, the following attributes are captured: i) target, ii) defect type, iii) qualifier, iv) source, and v) age.

Although ODC includes several attributes, we focus on two of them: defect type and qualifier. We do not consider the other ones as they do not bring relevant information towards identifying potential improvements to vulnerability detection (e.g., *activity* characterizes the moment that the vulnerability was discovered, but we are focused on how it was fixed and not when). We use the definition of defect type from [45], and the possible values are:

1) **Assignment/Initialization**: a problem related to an assignment of a variable or no assignment at all;
2) **Checking**: a problem with conditional logic (e.g., condition in a if-clause or in a loop);
3) **Timing**: a problem with serialization of shared resources;
4) **Algorithm/Method**: a problem with implementation that does not require a design change to be fixed;
5) **Function**: a problem that needs a reasonable amount of code to be fixed due to incorrect implementation or no implementation at all;
6) **Interface**: a problem in the interaction between components (e.g., parameter list).

[3]https://github.com

As for the qualifier, we can have:

1) **Missing**: new code needs to be added to fix the defect;
2) **Incorrect**: the code is incorrectly implemented and needs adjustment to fix the defect;
3) **Extraneous**: unnecessary code needs to be removed to fix the defect.

As GitHub highlights the code changes from the vulnerable to the neutral version, we have the information needed to classify these two attributes for each vulnerability. However, because the classification has to be done manually, we decided to have it done by two different researchers to get more accurate results. To have a common baseline for the two researchers, they started by analyzing and classifying together a sub-set of 30 vulnerabilities (out of the 159), which allowed discussing divergences in the approach and reaching a common rationale. This is very important as the experience of the two researchers is different. While one is a post-doctoral researcher that has worked with ODC before, the other is a Ph.D. student having the first practical experience with ODC in this work. After that step, each researcher classified the remaining 129 vulnerabilities individually, and the results were merged and consolidated at the end (the divergences in the classification were discussed to come up with a final classification).

As the fix of some vulnerabilities involves several code changes (in more than one block of code in a file or even in several files), each vulnerability may lead to a different number of ODC classifications by different researchers. Hence, the total number of ODC classifications (as presented in section IV) is larger than the total of vulnerabilities analyzed (Table 1).

To assess if the classification of the researchers is consistent, we calculated the inter-rater reliability (IRR) metric Cohen's Kappa [46] on the items classified separately by the two researchers. To interpret the metric, we use the *Landis and Koch* interpretation [47]: *a)* less than 0: no agreement; *b)* 0–0.20: slight agreement; *c)* 0.21–0.40: fair agreement; *d)* 0.41–0.60: moderate agreement; *e)* 0.61–0.80: substantial agreement; and *f)* 0.81–1.0 almost perfect.

E. COLLECT AND ANALYZE SAT ALERTS

To collect the SAT alerts, we ran two widely used SATs in both the vulnerable and neutral code versions for each vulnerability. The tools used were **CppCheck** (version 1.82) [48], a widely-used open-source SAT for C/C++, and **Flawfinder** (version 2.0.10) [49], another open-source SAT for C/C++ developed by D. A. Wheeler. Although there are other available C/C++ SATs (such as the ones used in [32]), they are either commercial or require the source code to be compiled. Examples of studies that use CppCheck or Flawfinder include [32], [39], [50]–[52]. Also, these SATs are known for revealing vulnerabilities, such as CVE-2017-1000249, which was initially identified by CppCheck. As the vulnerabilities are from a prolonged period (from 2000 to 2016) and exist in different versions of the codebase, it would not be possible to automate the SAT alert collection (even in the versions of

the same project) with the selected SATs. Moreover, some projects changed the build mechanism, such as Mozilla, which stopped using `make` and started using `mach`.

By analyzing the differences in terms of the alerts reported in the two versions of each code unit, we can study the ones that disappear due to a vulnerability fix (when compared with the vulnerable version) and the ones that appear in a vulnerability fix (which can lead to other vulnerabilities in the future).

F. COLLECT AND ANALYZE SOFTWARE METRICS

SMs can be obtained using tools such as SciTools Understand [44], and can be calculated at several levels, including files, functions, and classes. In fact, as mentioned before, a large number of SMs has been collected and made available by *Alves et al. [11], [35]* for five different projects. From that dataset, we used the file-level metrics for the Linux Kernel, Mozilla, and Xen projects (for the 159 buffer overflow vulnerabilities). We considered only the SMs at the file level, as most of the C/C++ code in these projects is not structured in classes. Also, for our analysis, the function metrics would not provide more information compared to file metrics, as some files are not structured in functions (such as scripts or header files). Hence, we decided to keep in the higher-level SMs of files. The main SMs are detailed in Appendix.

IV. MAIN CODE CHANGES WHEN FIXING VULNERABILITIES (RQ1)

The 159 buffer overflow vulnerabilities were classified according to the two ODC attributes, defect type and qualifier, by the two researchers. The subset of 30 vulnerabilities classified by the two researchers together (in a meeting that lasted a bit more than one hour) led to a total of 35 classifications, as some vulnerabilities were classified with more than a pair of type/qualifier values. Although ODC is an orthogonal classification (meaning that an attribute should not have more than one value assigned), this happens in our case as a vulnerability fix may require one or more independent code changes. In other words, several block changes may be needed in a single fix, which leads to more than one ODC classification per vulnerability (in practice, we can say that several code weaknesses/faults lead to one vulnerability).

The 129 vulnerabilities that were classified separately resulted in 167 and 177 classifications by each researcher (each one spent between 5 to 6 hours in this classification step). With these data, we computed the IRR Cohen's Kappa metric to assess the consistency between the classifications. We obtained the following values: *i) defect type*: 0.4476 (moderate agreement), and *ii) qualifier*: 0.3939 (fair agreement). Additionally, we computed the Cohen's Kappa considering both ODC attributes (*defect type + qualifier*) as a single classification. The result is 0.4255, a moderate agreement between the researchers. Although these results do not indicate an excellent agreement, they show a fair to moderate agreement according to *Landis and Koch* interpretation [47],

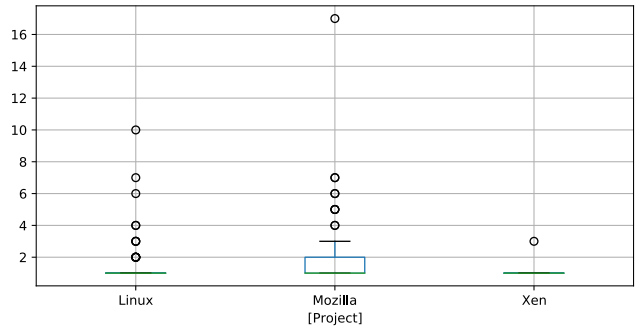

FIGURE 2. Number of changed files per vulnerability fix.

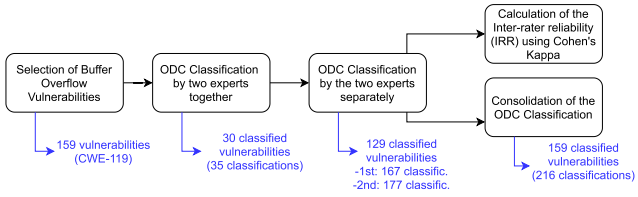

FIGURE 3. Summary of the analysis of the buffer overflow vulnerabilities using ODC.

suggesting that our ODC classification for the buffer overflow vulnerabilities is quite consistent.

The divergences in the classifications were discussed (in a meeting that lasted 2 hours), resulting in 216 classifications (pairs of defect types and qualifiers) for the 159 vulnerabilities. This happens because 35 (22.01%) vulnerabilities were fixed by changing more than one file, as can be seen in the box plot in Figure 2, which indicates that some buffer overflow vulnerabilities are due to the interaction of more than one software component. Such interactions clearly make the vulnerability detection process difficult to automate.

A summary of the ODC classification process can be seen in Figure 3 and the consolidated results are presented in Table 2. Regarding the defect type, the most frequent vulnerabilities are from "Checking" (85 cases, 39.35%), followed by "Algorithm/Method" (64 cases, 29.63%), and "Assignment/Initialization" (42 cases, 19.44%). This confirms the observations of Morrison *et al.* [36] in their study using ODC+V. Regarding the qualifier, "Incorrect" is the most frequent one (123 cases, 56.94%) followed by "Missing" (88 cases, 40.74%).

Table 3 (defect type) and Table 4 (qualifier) summarize the results of classification per project. Similar to the overall results, the majority of the classifications belong to the "Checking" defect type, for both Linux Kernel and Mozilla projects. This seems to be an obvious approach to prevent out-of-bound access, but developers still fail to add them to the source code. This probably happens as the developers do not anticipate the need for the "Checking" since it is difficult to consider all possibilities and identify the ones that may lead to a buffer overflow problem. Moreover, the developers are probably not supported by adequate tools that help them iden-

TABLE 2. Vulnerability distribution across the ODC defect type and the qualifiers.

Defect Type	Incorrect	Missing	Extraneous	Total
Checking	31 (14.35%)	51 (23.61%)	3 (1.39%)	85 (39.35%)
Algorithm/Method	50 (23.15%)	12 (5.56%)	2 (0.93%)	64 (29.63%)
Assignment/Initialization	24 (11.11%)	18 (8.33%)	0 (0.00%)	42 (19.44%)
Interface	16 (7.41%)	3 (1.39%)	0 (0.00%)	19 (8.80%)
Function	2 (0.93%)	4 (1.85%)	0 (0.00%)	6 (2.78%)
Timing	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	123 (56.94%)	88 (40.74%)	5 (2.31%)	216 (100.00%)

TABLE 3. Vulnerability distribution across the ODC defect type for the projects (Linux Kernel, Mozilla, Xen).

Defect Type	Linux Kernel	Mozilla	Xen
Checking	45 (37.50%)	39 (42.86%)	1 (20.00%)
Algorithm/Method	33 (27.50%)	27 (29.67%)	4 (80.00%)
Assignment/Initialization	27 (22.50%)	15 (16.48%)	0 (0.00%)
Interface	11 (9.17%)	8 (8.79%)	0 (0.00%)
Function	4 (3.33%)	2 (2.20%)	0 (0.00%)
Timing	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	120 (100.00%)	91 (100.00%)	5 (100.00%)

TABLE 4. Vulnerability distribution across the ODC qualifier for the projects (Linux Kernel, Mozilla, Xen).

Qualifier	Linux Kernel	Mozilla	Xen
Incorrect	74 (61.67%)	46 (50.55%)	3 (60.00%)
Missing	45 (37.50%)	42 (46.15%)	1 (20.00%)
Extraneous	1 (0.83%)	3 (3.30%)	1 (20.00%)
Total	120 (100.00%)	91 (100.00%)	5 (100.00%)

tifying or verifying the "Checking" and conditions. Another possible reason is the lack of testing skills, such as applying boundary-value analysis when creating the functionality.

Regarding the qualifier attribute, more than half of the Linux Kernel classifications are labeled as "Incorrect" (74 cases, 61.67%), while for the Mozilla project the number of "Incorrect" (46 cases, 50.55%) and "Missing" (42 cases, 46.15%) are quite similar. The small number of classifications for the Xen project does not allow further analysis.

The above results help answering the **RQ1**, about *the main changes in the code when fixing buffer overflow vulnerabilities*, and indicate that projects lack mechanisms to verify simple conditional logic ("Checking" defect type) and assignment and initialization of variables ("Assignment/Initialization" defect type). Moreover, some review effort could be beneficial to identify major issues in the implementation ("Algorithm/Method" defect type). In all cases, issues can be either absent ("Missing" qualifier) or poorly implemented ("Incorrect" qualifier) code. At first sight, the detection of some of these cases may be easily automated (e.g., by adding simple rules to SATs), but many others require human intervention or other advanced detection techniques (e.g., in the case of vulnerabilities that occur due to multiple weaknesses in different files). As an example, let's take a look at the Linux Kernel vulnerability CVE-2014-0049, whose fix can be seen below. It shows the code of the Linux Kernel file `arch/x86/kvm/x86.c` after the fix of CVE-2014-0049.

```
if (vcpu->mmio_cur_fragment >= vcpu->
  mmio_nr_fragments) {
  // removed due to space constraints
}
```

In this example, the *if-clause* was classified as being "Incorrect" because the operator had to be changed from == (equal comparison in C/C++) to >= (greater than or equal comparison) to fix the vulnerability. The buffer overflow happened as the execution of the function was not interrupted when the number of the current fragment exceeded the total number of fragments. This cannot be automated through SAT rules as it requires a complex and semantic interpretation. Although the Linux Kernel development team could create a rule for this specific case, it turns to be totally context-specific, thus, when used in different contexts, it would lead to a high number of false alarms.

Key Observations:

- Most buffer overflow vulnerabilities are fixed by a simple change in conditional logic (either incorrect or missing), which is not anticipated by developers and not identified by tools
- Some vulnerabilities could have been detected with the use of adequate SATs
- Not all vulnerabilities can have their identification automated as they involve intricate issues in several files
- A manual review process could help identifying vulnerabilities earlier in the SDLC

V. SAT ALERTS BEFORE AND AFTER VULNERABILITY FIXES (RQ2)

The analysis of the SAT alerts started with the execution of two tools (CppCheck and Flawfinder) over the two versions

TABLE 5. Number of alerts (minimum and maximum among all commits) reported by the SATs for the complete code-base in all vulnerabilities.

	CppCheck	Flawfinder
Linux Kernel	35, 110 (min.)	32, 157 (min.)
	79, 135 (max.)	55, 930 (max.)
Mozilla	24, 660 (min.)	5, 755 (min.)
	45, 769 (max.)	22, 543 (max.)
Xen	4, 545 (min.)	3, 079 (min.)
	4, 717 (max.)	3, 216 (max.)

of each code (vulnerable and neutral) for each buffer overflow vulnerability. With this, we can analyze what has been changed between the vulnerable and the neutral version. Three cases are possible: *i)* alerts disappearing from the vulnerable version to the neutral version due to the code fix; *ii)* new alerts appearing in the code that fixed the vulnerability; and *iii)* SAT alerts either appearing or disappearing in untouched code (part of the code that was not affected by the fixes). Depending on the techniques used by the SATs (e.g., data flow analysis, taint analysis), some code changes may cause raising new alerts in parts of the code that were not touched. Thus, we consider all the alerts, and not only the ones in the changed code. Nevertheless, as most alerts are kept equal from one version to the other, we can filter these out. In practice, we are simplifying the analysis by excluding the alerts that have the same type and are raised in the same (or corresponding) lines of code in both the vulnerable and the neutral versions of each code unit.

Considering the types of SAT alerts for which we identified variations between the vulnerable and the neutral versions of each vulnerability, none appeared in more than one project. For example, the alert type `wcscpy` from Flawfinder is raised in the Mozilla project but not in the Linux Kernel and Xen projects. This happens both for CppCheck and Flawfinder alerts in the different projects. Due to the reduced number of vulnerabilities in Xen, we could not observe much regarding SAT alerts changing due to code fixes: only one alert with CppCheck (`doubleFree`) and none with Flawfinder.

Table 5 shows a summary of the minimum and the maximum number of alerts per SAT raised for different commits of each project. For example, Linux Kernel has 98 vulnerabilities analyzed in this study (see Table 1), we ran the SATs for those vulnerabilities for different commits, and counted all reported alerts per vulnerability for each commit. The smallest number of alerts raised for Linux Kernel by CppCheck belongs to a specific commit and is equal to 35, 110 alerts, and the largest number of alerts raised for Linux Kernel by CppCheck belongs to another commit and is equal to 79, 135. Overall, CppCheck reports more alerts for the complete codebase of each project than Flawfinder. The project with the largest number of alerts is Linux Kernel (maximum of 79, 000 reported by CppCheck). On the other hand, Xen is the project with the smallest number of reported alerts (minimum of 3, 000 reported by Flawfinder).

TABLE 6. CppCheck SAT alerts reported in Linux Kernel that changed from the vulnerable to the neutral versions.

	Linux Kernel	
SAT	Vulnerable	Neutral
nullPointerRedundantCheck	3	0
unusedStructMember	1	0
Total	4	0

TABLE 7. CppCheck SAT alerts reported in Mozilla that changed from the vulnerable to the neutral versions.

	Mozilla	
SAT	Vulnerable	Neutral
syntaxError	1	1
toomanyconfigs	0	1
uninitMemberVar	1	0
unusedFunction	1	0
memsetClassFloat	1	0
Total	4	2

TABLE 8. Flawfinder SAT alerts reported in Linux Kernel that changed from the vulnerable to the neutral versions.

		Linux Kernel	
Category	SAT Type	Vulnerable	Neutral
Buffer	memcpy	8	5
	char	6	6
	strlen	7	4
	strcpy	6	3
	sprintf	6	1
	strncpy	4	3
	strncat	1	0
Format	syslog	14	14
	printf	0	8
	vsnprintf	1	1
Total		53	45

TABLE 9. Flawfinder SAT alerts reported in Mozilla that changed from the vulnerable to the neutral versions.

		Mozilla	
Category	SAT Type	Vulnerable	Neutral
Buffer	wcscpy	19	0
	wcsncpy	0	19
	wcslen	1	0
Format	fprintf	1	0
Total		21	19

Table 6 and Table 7 show the CppCheck results for Linux Kernel and Mozilla, respectively. Table 8 and Table 9 show the Flawfinder results (as this tool defines categories for the types of alerts, they are also included in the tables). As shown, the total number of alerts that vary from vulnerable to neutral versions is very small when compared to the number of alerts

raised per project. For example, for the Mozilla project, the total number of Flawfinder alerts varies between 5, 755 and 22, 543 (Table 5) over the 56 commits considered. However, the alerts that changed between the vulnerable and neutral versions of all vulnerabilities are only 21 (Table 9). In other words, 21 alerts that were raised in vulnerable versions disappeared when fixes were implemented. On the other hand, 19 new alerts appeared after vulnerabilities fixes.

From the 159 vulnerabilities analyzed, only 22 fixes (13.84%) impacted the SAT alerts, but only one impacted the alerts raised by both SATs (CVE-2007-6151). Furthermore, although all projects are written in the same programming languages (C/C++) and the vulnerabilities analyzed are of the same type (CWE-119), very different SAT alerts are raised in the three projects (with no clear overlap), suggesting that the root causes of buffer overflow vulnerabilities differ a lot from each other. Let's analyze a couple of examples.

The following code snippet presents the fix of the Linux Kernel CVE-2010-4527 vulnerability in the file `linux/sound/oss/soundcard.c`. In this case, the vulnerable function `strcpy` has been replaced by `strncpy`, which is also considered vulnerable (line 102). Flawfinder raised alerts in both the vulnerable and the (supposedly) neutral versions, meaning that part of the alerts on the `strcpy` version migrated to `strncpy` version. This fix also changes the `strcmp` (considered unsafe) to `strncmp` (line 90), but none of the SATs raised an alert for th unsafe function `strcmp`.

```
// Line 90
if (strncmp(name, mixer_vols[i].name, 32) == 0)
  {

// Unchanged lines: 91-101

// Line 102
strncpy(mixer_vols[n].name, name, 32);
```

To replace the vulnerable function `strncpy` by a safe one, `strlcpy` can be used, as done in CVE-2013-2850 (file `drivers/target/iscsi/iscsi_target_parameters.c`). Note that the use of vulnerable functions is a known weakness still present in many software systems. In fact, OWASP lists "using components with known vulnerabilities" as one of the top 10 weaknesses that software developers should prevent [53].

```
strlcpy(extra_response->key, key, sizeof(
  extra_response->key));
```

The following code snippet shows an example of a "Checking" that was missing in the code, and that was not detected by any of the SATs. This is part of the vulnerability CVE-2013-1721. In the vulnerable version, only the first part of the condition was included; the second condition has been added to fix the vulnerability by checking the required space needed for a buffer in use.

```
  else if (mWritePosition + requiredSpace >
  mBufferSize ||
          mWritePosition + requiredSpace <
  mWritePosition)   // Recycle
```

These results help answering **RQ2** about *the difference between SAT alerts reported before and after fixes*. As shown, in most cases, the vulnerability fixes do not change the outcome of the SATs, suggesting a low capability of these tools to detect buffer overflow vulnerabilities.

Key Observations:

- A small number of vulnerabilities are detected by SATs, especially when unsafe functions are used
- Some fixes lead to new SAT alerts, sometimes related to the use of other unsafe functions (e.g., `wcscpy` to `wcsncpy`)
- Not all vulnerabilities can be detected by SATs as they involve the interaction among diverse components
- New SAT rules are needed to detect specific vulnerabilities, in particular the other related to *checking* conditions

VI. IMPACT ON SOFTWARE METRICS (SMs) DUE TO VULNERABILITY FIXES (RQ3)

One of the main arguments in previous works for using SMs to detect software vulnerabilities is that they allow portraying the size and complexity of the source code and, usually, more complex code is more prone to have vulnerabilities [9], [10]. However, this assumption needs to be confirmed in order to gain trust in the use of SMs to detect software vulnerabilities. The 54 SMs used in this study portray different characteristics of the source code, such as volume, coupling, cohesion, and complexity.

Table 10 presents the top 10 of the SMs that changed more frequently when buffer overflow vulnerabilities were fixed, which are mostly volume metrics. For example, considering the Linux Kernel project, we can observe that the *CountLine* metric changed for 68 files, out of a total of 75 files modified to fix 98 vulnerabilities. The *CountLine* metric indicates the number of physical lines in a file. As shown, nine out of the ten metrics are the same for the three projects. They either reflect the size of the code (*CountLine*, *AltCountLineCode*, *CountLineCode*, *CountStmt*, *CountLineCodeExe*, *CountSemicolon*, and *CountStmtExe*) or its complexity (*SumCyclomaticStrict* and *SumCyclomaticModified*). Table 12 in Appendix details all the SMs presented in this section.

Although this suggests a correlation between vulnerability fixes and the value of some metrics, there may not be causality. In fact, a detailed analysis of the values of volume (e.g., lines of code) and complexity (e.g., McCabe cyclomatic complexity) metrics allows observing that the value of most metrics increases when a buffer overflow vulnerability is fixed. This is confirmed by the ODC analysis, in which 40.74% of the classifications showed that some code had to be added to fix a vulnerability. The problem is that this may

TABLE 10. Top 10 SMs impacted by the vulnerability fixes per project (more than 10 items listed as Xen ties in some SMs).

SM	Linux Kernel	Mozilla	Xen
CountLine	68/75 (90.7%)	44/51 (86.3%)	6/6 (100.0%)
AltCountLineCode	66/75 (88.0%)	43/51 (84.3%)	6/6 (100.0%)
CountLineCode	59/75 (78.7%)	38/51 (74.5%)	5/6 (83.3%)
CountStmt	56/75 (74.7%)	34/51 (66.7%)	5/6 (83.3%)
CountStmtExe	53/75 (70.7%)	27/51 (52.9%)	5/6 (83.3%)
CountSemicolon	53/75 (70.7%)	32/51 (62.7%)	5/6 (83.3%)
CountLineCodeExe	53/75 (70.7%)	33/51 (64.7%)	5/6 (83.3%)
HK	49/75 (65.3%)	-	4/6 (66.7%)
SumCyclomaticStrict	46/75 (61.3%)	24/51 (47.1%)	4/6 (66.7%)
SumCyclomaticModified	42/75 (56.0%)	23/51 (45.1%)	4/6 (66.7%)
SumCyclomatic	-	23/51 (45.1%)	4/6 (66.7%)
AltCountLineBlank	-	-	4/6 (66.7%)
CountLineBlank	-	-	4/6 (66.7%)

TABLE 11. Unchanged SMs in the vulnerability fixes per project.

Project	Unchanged SMs
Linux Kernel	*CountDeclClass*, DIT, NOC, CBC, RFC, CBO, LCOM
Mozilla	MaxNesting, CountStmtEmpty, *CountDeclClass*, AvgLineBlank, AltAvgLineBlank
Xen	NOC, DIT, CBC, RFC, CBO, AltAvgLineBlank, RatioCommentToCode, MaxNesting, CountStmtEmpty, CountLinePreprocessor, CountDeclFunction, *CountDeclClass*, AvgLineComment, AvgLineCode, AvgLineBlank, AvgLine, AvgEssential, AvgCyclomaticStrict, AvgCyclomaticModified, AvgCyclomatic, AltAvgLineComment, LCOM

go against the assumption that smaller and simpler code is less prone to be vulnerable.

Moreover, although the value of some metrics is frequently varying from the vulnerable to the neutral versions, it does not say much about the potential existence of vulnerabilities. For example, the metric *Henry Kafura Size (HK)*, which was proposed by Henry and Kafura [54], also appears as being frequently impacted by vulnerability fixes (in fact, it is among the metrics whose values varies more). HK is the result of the multiplication of three other metrics, being two of them squared ($length*(FanIn*FanOut)^2$, where *length* is a volume metric, such as LOC). Consequently, small changes in the code can result in a large value variation for *HK*, but we cannot assure that is not strictly related to vulnerability fixes (in fact, we can observe similar variations across different versions of the same file, even when no vulnerabilities are being dealt with).

Table 11 shows the SMs that remain unchanged due to vulnerability fixes per project. For example, the SM *MaxNesting* has the same value before and after fixing the vulnerability in all the analyzed files for Mozilla and Xen projects. Among these, there is one, *CountDeclClass* (number of classes), that never varies at all (in italic in the table). Clearly, these are metrics that cannot help in the detection of software vulnerabilities.

These results help answering **RQ3** about *SMs changing when buffer overflow vulnerabilities are fixed*. Although most of the fixes lead to changes in SMs, metrics are not more impacted by fixes than by other code improvements. In fact, there is no clear causality between the value of a metric and the existence of a vulnerability. Hence, no SM can be used to detect the presence of buffer overflow vulnerabilities. This has been confirmed by other works that use SMs to detect software vulnerabilities [14], [55].

Key Observations:

- Most vulnerability fixes add code to the codebase, leading to an increase in the value of the metrics that are related to volume and complexity
- Causality between vulnerability fixes and the variation on metrics cannot be established
- Code changes related to vulnerability fixes cannot be easily distinguished from other improvements using software metrics
- SMs are not good indicators of the existence of vulnerabilities, in particular buffer overflow, but probably can be used to indicate less trustworthy code units

VII. THREATS TO VALIDITY

This section discusses the threats to the validity of the approach and the results obtained. The threats are mainly related to the projects considered, the ODC classification, the SATs used, and the number and types of vulnerabilities analyzed.

The study considers a single type of vulnerability: buffer overflow. Although it is the most relevant vulnerability type for C/C++ projects, other vulnerabilities (e.g., input validation) are not considered and may lead to different observations and conclusions as the SAT alerts and SMs may vary.

All the projects analyzed in this study are developed in C/C++ and have a large code-base. Buffer overflow vulnerabilities are more relevant for C/C++ projects, but not limited to them. Hence, some key observations may not be the same for projects in other programming languages and with different sizes. Nevertheless, the observations are still relevant as buffer overflow is still one of the most frequent issues in many programming languages and has a severe impact on software security, particularly in C/C++ projects that compose many essential and highly used software projects.

The different levels of experience of the researchers and the complexity of some of the fixes may lead to different classification results. We tried to mitigate this by performing an initial joint classification effort before the individual ones. Moreover, individual classifications were discussed in a consensus meeting to reach a final agreed classifi-

TABLE 12. Description of the Software Metrics at the File level (adapted from SciTools understand (https://support.scitools.com/t/what-metrics-does-understand-have/66)).

Software Metric	Name	Description
AltAvgLineBlank	Average Number of Blank Lines (Include Inactive)	Average number of blank lines for all nested functions or methods, including inactive regions.
AltAvgLineCode	Average Number of Lines of Code (Include Inactive)	Average number of lines containing source code for all nested functions or methods, including inactive regions.
AltAvgLineComment	Average Number of Lines with Comments (Include Inactive)	Average number of lines containing comment for all nested functions or methods, including inactive regions.
AltCountLineBlank	Blank Lines of Code (Include Inactive)	Number of blank lines, including inactive regions.
AltCountLineCode	Lines of Code (Include Inactive)	Number of lines containing source code, including inactive regions.
AltCountLineComment	Lines with Comments (Include Inactive)	Number of lines containing comment, including inactive regions.
AvgCyclomatic	Average Cyclomatic Complexity	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	Average Modified Cyclomatic Complexity	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	Average Strict Cyclomatic Complexity	Average strict cyclomatic complexity for all nested functions or methods.
AvgEssential	Average Essential Cyclomatic Complexity	Average Essential complexity for all nested functions or methods.
AvgFanIn	Average Inputs	Average of the number of calling subprograms plus global variables read.
AvgFanOut	Average Output	Average of the number of called subprograms plus global variables set.
AvgLine	Average Number of Lines	Average number of lines for all nested functions or methods.
AvgLineBlank	Average Number of Blank Lines	Average number of blank for all nested functions or methods.
AvgLineCode	Average Number of Lines of Code	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	Average Number of Lines with Comments	Average number of lines containing comment for all nested functions or methods.
AvgMaxNesting	Average Nesting	Average nesting level of control constructs.
CBC (CountClassBase)	Base Classes	Number of immediate base classes. (aka IFANIN)
CBO (CountClassCoupled)	Coupling Between Objects	Number of other classes coupled to.
CountDeclClass	Classes	Number of classes.
CountDeclFunction	Function	Number of functions.
CountLine	Physical Lines	Number of all lines. (aka NL)
CountLineBlank	Blank Lines of Code	Number of blank lines. (aka BLOC)
CountLineCode	Source Lines of Code	Number of lines containing source code. (aka LOC)
CountLineCodeDecl	Declarative Lines of Code	Number of lines containing declarative source code.
CountLineCodeExe	Executable Lines of Code	Number of lines containing executable source code.
CountLineComment	Lines with Comments	Number of lines containing comment. (aka CLOC)
CountLineInactive	Inactive Lines	Number of inactive lines.
CountLinePreprocessor	Preprocessor Lines	Number of preprocessor lines.
CountPath	Paths	Number of possible paths, not counting abnormal exits or gotos. (aka NPATH)
CountSemicolon	Semicolons	Number of semicolons.
CountStmt	Statements	Number of statements.
CountStmtDecl	Declarative Statements	Number of declarative statements.
CountStmtEmpty	Empty Statements	Number of empty statements.
CountStmtExe	Executable Statements	Number of executable statements.
DIT (MaxInheritanceTree)	Depth of Inheritance Tree	Maximum depth of class in inheritance tree.
FanIn	Inputs	Number of calling subprograms plus global variables read. (aka CountInput)
FanOut	Outputs	Number of called subprograms plus global variables set. (aka CountOutput)
HK	Henry Kafura Size [54]	Interconnectivity of a procedure with its environment.
LCOM (PercentLackOfCohesion)	Lack of Cohesion in Methods	100% minus the average cohesion for package entities. (aka LCOM, LOCM)
MaxCyclomatic	Max Cyclomatic Complexity	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	Max Modified Cyclomatic Complexity	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Max Strict Cyclomatic Complexity	Maximum strict cyclomatic complexity of nested functions or methods.
MaxEssential	Max Essential Complexity	Maximum essential complexity of all nested functions or methods.
MaxMaxNesting	Max Nesting	Maximum of Maximum nesting level of control constructs.
MaxNesting	Nesting	Maximum nesting level of control constructs.
NOC (CountClassDerived)	Number of Children	Number of immediate subclasses.
RatioCommentToCode	Comment to Code Ratio	Ratio of comment lines to code lines.
RFC (CountDeclMethodAll)	Methods	Number of methods, including inherited ones. (aka RFC: response for class)
SumCyclomatic	Sum Cyclomatic Complexity	Sum of cyclomatic complexity of all nested functions or methods. (aka WMC)
SumCyclomaticModified	Sum Modified Cyclomatic Complexity	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	Sum Strict Cyclomatic Complexity	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	Sum Essential Complexity	Sum of essential complexity of all nested functions or methods.
SumMaxNesting	Sum Max Nesting	Sum of maximum nesting level of control constructs.

cation. However, both researchers may have misclassified, in the same way, some vulnerabilities. To mitigate this issue in the future, more experts can be asked to perform the classification.

The number of SATs used to generate alerts is limited to two. Nevertheless, these two SATs (CppCheck and Flawfinder) are widely used and have a high number of rules to detect software issues and vulnerabilities. Hence, they provide relevant input for the analysis.

Although the main objective of the selected tools for this study is to detect vulnerabilities, we used them to characterize only buffer overflow vulnerabilities. Thus, another limitation of the study is related to the limited techniques used for the characterization of the buffer overflow vulnerabilities, as other software vulnerability detection techniques, in particular, dynamic techniques such as SPT and fuzzing, could reveal additional characteristics of the buffer overflow vulnerabilities. Nevertheless, due to the project characteristics and the time span of the vulnerabilities, it would not be doable to apply techniques like that in the dataset considered for this study.

Only 159 vulnerabilities are analyzed. This is not a large number, but the diversity of the causes and the representativeness of the projects support some relevant observations. Moreover, most of the analysis involves manual validation of the vulnerability fixes. Hence, performing this study in a larger dataset would be even more time-consuming. Automatizing the ODC classification would be an option (such as it was performed in [56]). However, a manual review would still be needed to validate the classification and fix the incorrect ones.

VIII. CONCLUSION AND FUTURE WORK

This work analyzed a set of buffer overflow vulnerabilities to study potential ways to improve vulnerability detection, either by improving existing techniques or devising new ones. The vulnerabilities of three open-source C/C++ projects were used in the analysis (Linux Kernel, Mozilla, and Xen). Each vulnerability was classified using ODC. Moreover, the SAT alerts and SMs were analyzed and compared for both the vulnerable and neutral versions. The results showed that most of the vulnerable code units are labeled with ODCs defect types *checking* and *algorithm/method*. On the other hand, SATs lack rules to detect most vulnerabilities, in particular missing or incorrect checking logic. Also, we could not find any causality between buffer overflow vulnerabilities fixes and the value of SMs.

As future work, we plan to expand the analysis for other types of vulnerability as well as considering more recent vulnerabilities. The results of this work will help us to develop new SAT rules or improve the existing ones to trigger alerts that allow discovering more software vulnerabilities. The goal is to plug the new or improved rules into an existing SAT, such as CppCheck or Flawfinder. Moreover, we plan to create a prioritization mechanism to sort the SAT alerts to be analyzed by the software development teams reducing the review and correction cost. Also, we intend to understand the differences between buffer overflow vulnerabilities and other vulnerabilities types from the point of view of SAT alerts and SMs. Finally, we plan to conduct a study to understand the changes in SAT alerts and SMs in commits that introduced vulnerabilities, and also at different levels of the source code (*e.g.*, modules or functions).

APPENDIX
SOFTWARE METRICS

SMs are presented in Table 12, with their name and description based on the SciTools Understand.

REFERENCES

[1] *Information Technology—Security Techniques—Information Security Management Systems—Overview and Vocabulary*, Standard, International Organization for Standardization, Geneva, Switzerland, Standard ISO/IEC 27000:2018, Feb. 2018.
[2] L. Jaffee. (2020). *COVID-19 Accounts For Most 2020 Cyberattacks*. Accessed: May 3, 2021. [Online]. Available: https://www.scmagazine.com/home/security-news/covid-19-accounts-for-most-2020-cyberattacks/
[3] S. Zurier. (2021). *Security Spending Will Top 40% in Most 2021 IT Budgets*. Accessed: May 3, 2021. [Online]. Available: https://www.scmagazine.com/home/security-news/security-spending-will-top-40-in-most-2021-it-budgets/
[4] White Source. (2021). *What are the Most Secure Programming Languages?*. Accessed: Mar. 9, 2021. [Online]. Available: https://www.whitesourcesoftware.com/most-secure-programming-languages/
[5] K. Turpin. (2010). *OWASP Secure Coding Practices—Quick Reference Guide*. Accessed: Jun. 20, 2019. [Online]. Available: https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf
[6] Software Engineering Institute—Carnegie Mellon University. *SEI CERT C++ Coding Standard*. Accessed: May 11, 2021. [Online]. Available: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682
[7] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.
[8] B. Chess and J. West, *Secure Programming With Static Analysis*, 1st ed. Reading, MA, USA: Addison-Wesley, 2007.
[9] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 23–33.
[10] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Dec. 2011.
[11] H. Alves, B. Fonseca, and N. Antunes, "Experimenting machine learning techniques to predict vulnerabilities," in *Proc. 7th Latin-Amer. Symp. Dependable Comput. (LADC)*, Oct. 2016, pp. 151–156.
[12] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software metrics as indicators of security vulnerabilities," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2017, pp. 216–227.
[13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA, May 2013, pp. 672–681.
[14] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Vulnerable code detection using software metrics and machine learning," *IEEE Access*, vol. 8, pp. 219174–219198, 2020.
[15] K. Kratkiewicz and R. Lippmann, "Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools," in *Proc. Workshop Eval. Softw. Defect Detection Tools*, 2005, p. 19.
[16] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *Proc. 4th Int. Workshop Predictor Models Softw. Eng. (PROMISE)*, New York, NY, USA, 2008, pp. 11–18.
[17] N. Neves, J. Antunes, M. Correia, P. Verissimo, and R. Neves, "Using attack injection to discover new vulnerabilities," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2006, pp. 457–466.
[18] (2006). *CWE-119: Improper Restriction of Operations With in the Bounds of a Memory Buffer*. Accessed: May 3, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/119.html
[19] MITRE. *Common Weakness Enumeration*. Accessed: May 3, 2021. [Online]. Available: https://cwe.mitre.org
[20] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *Proc. 4th Int. Conf. Multimedia Inf. Netw. Secur.*, Nov. 2012, pp. 152–156.
[21] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Secur. Privacy*, vol. 3, no. 1, pp. 84–87, Jan. 2005.
[22] B. Chess and G. McGraw, "Static analysis for security," *IEEE Secur. Privacy*, vol. 2, no. 6, pp. 76–79, Nov. 2004.

[23] *SpotBugs*. Accessed: May 5, 2021. [Online]. Available: https://spotbugs.github.io
[24] J. Kresowaty, "FxCop and code analysis: Writing your own custom rules," Tech. Rep., 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.466.282&rep=rep1&type=pdf
[25] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, May 2006, p. 263.
[26] Parasoft. *Parasoft CPPTest—C/C++ Static Code Analysis*. Accessed: May 11, 2021. [Online]. Available:
[27] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Shelter Island, NY, USA: Manning Publications, 2013.
[28] Synopsys. *Coverity Static Application Security Testing*. Accessed: May 11, 2021. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
[29] W. R. J. Freitez, A. Mammar, and A. R. Cavalli, "Software vulnerabilities, prevention and detection methods: A review," in *SEC-MDA: Security in Model Driven Architecture*. Enschede, The Netherlands: Springer, 2009, pp. 1–11.
[30] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications*, D. Mandal, R. Kar, S. Das, and B. K. Panigrahi, Eds. New Delhi, India: Springer, 2015, pp. 581–591.
[31] A. H. Watson, D. R. Wallace, and T. J. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," US Dept. Commerce, Technol. Admin., Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. 500, 1996.
[32] A. Arusoaie, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in C/C++ code," in *Proc. 19th Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Sep. 2017, pp. 161–168.
[33] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Nov. 2015, pp. 12–15.
[34] Y. Nong, H. Cai, P. Ye, L. Li, and F. Chen, "Evaluating and comparing memory error vulnerability detectors," *Inf. Softw. Technol.*, vol. 137, Sep. 2021, Art. no. 106614.
[35] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: Dataset and exploratory study," in *Proc. 12th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2016, pp. 37–44.
[36] P. J. Morrison, R. Pandita, X. Xiao, R. Chillarege, and L. Williams, "Are vulnerabilities discovered and resolved like other defects?" *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1383–1421, 2018.
[37] A. Shostack, *Threat Modeling: Designing for Security*. Hoboken, NJ, USA: Wiley, 2014.
[38] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006.
[39] 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," 2018, *arXiv:1801.01681*. [Online]. Available: http://arxiv.org/abs/1801.01681
[40] National Institute of Standards and Technology (NIST). (2005). *National Vulnerability Database*. Accessed: Aug. 24, 2021. [Online]. Available: https://nvd.nist.gov
[41] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 989–1006.
[42] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proc. 22nd USENIX Secur. Symp. (USENIX Secur.)*, Washington, DC, USA, Aug. 2013, pp. 49–64.
[43] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, New York, NY, USA, Jun. 2020, pp. 1547–1559.
[44] SciTools. (2011). *SciTools Understand—Metrics*. Accessed: Apr. 3, 2020. [Online]. Available: https://scitools.com/feature/metrics/
[45] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification–A concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.
[46] J. Cohen, "A coefficient of agreement for nominal scales," *Educ. Psychol. Meas.*, vol. 20, no. 1, pp. 37–46, 1960.
[47] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.
[48] D. Marjamäki. *Cppcheck—A Tool for Static C/C++ Code Analysis, 2007*. Accessed: Aug. 30, 2019. [Online]. Available: http://cppcheck.sourceforge.net
[49] D. A. Wheeler. *Flawfinder, 2001*. Accessed: Aug. 30, 2019. [Online]. Available: https://dwheeler.com/flawfinder/
[50] L. R. Russell, Y. L. Kim, H. L. Hamilton, T. Lazovich, A. J. Harer, O. Ozdemir, M. P. Ellingwood, and W. M. McConley, "Automated vulnerability detection in source code using deep representation learning," 2018, *arXiv:1807.04320*. [Online]. Available: https://arxiv.org/abs/1807.04320
[51] M. Zhou, J. Chen, Y. Liu, H. Ackah-Arthur, S. Chen, Q. Zhang, and Z. Zeng, "A method for software vulnerability detection based on improved control flow graph," *Wuhan Univ. J. Natural Sci.*, vol. 24, no. 2, pp. 149–160, Apr. 2019.
[52] A. Al-Boghdady, K. Wassif, and M. El-Ramly, "The presence, trends, and causes of security vulnerabilities in operating systems of IoT's low-end devices," *Sensors*, vol. 21, no. 7, p. 2329, 2021.
[53] A. van der Stock, B. Glas, N. Smithline, and T. Gigler. (2017). *OWASP Top 10–2017–The Ten Most Critical Web Application Security Risks*. Accessed: May 4, 2021. [Online]. Available: https://owasp.org/www-project-top-ten/2017/
[54] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. SE-7, no. 5, pp. 510–518, Sep. 1981.
[55] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Softw. Eng.*, vol. 18, no. 1, pp. 25–59, 2013.
[56] F. Lopes, J. Agnelo, C. A. Teixeira, N. Laranjeiro, and J. Bernardino, "Automating orthogonal defect classification using machine learning algorithms," *Future Gener. Comput. Syst.*, vol. 102, pp. 932–947, Jan. 2020.

JOSÉ D'ABRUZZO PEREIRA (Member, IEEE) received the B.Sc. degree in computer science from the State University of Campinas (Unicamp), Brazil, and the M.Sc. degree in information technology and software engineering from the University of Coimbra and Carnegie Mellon University. He is currently pursuing the Ph.D. degree in information science and technology with the University of Coimbra (UC). He is also a member of the Software and System Engineering (SSE) Group, CISUC. His research interests include security and vulnerability detection, static code analysis, software project management, software quality, and self-adaptive systems.

NAGHMEH IVAKI (Member, IEEE) received the Ph.D. degree from the University of Coimbra, Portugal. She is currently a Researcher and a Full Member of the Centre for Informatics and Systems (CISUC), Software and Systems Engineering Group (SSE), Department of Informatics Engineering, University of Coimbra. She specializes in the scientific field of informatics engineering, with particular focus on security and dependability of computer systems. In her field of specialization, she has authored more than 30 peer-reviewed publications and participated in several national and international research projects.

MARCO VIEIRA (Member, IEEE) is currently a Full Professor with the University of Coimbra, Portugal. His research interests include dependability and security assessment and benchmarking, fault injection, software processes, and software quality assurance, subjects in which he has authored or coauthored more than 200 papers in refereed conferences and journals. He has participated and coordinated several research projects, both at the national and European level. He is also the Vice Chair of the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, an Associate Editor of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING journal, and has served as the Program Chair and on the program committees of the major conferences of the dependability area.