An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel

Matthieu Jimenez, Mike Papadakis, Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
Email: matthieu.jimenez@uni.lu, michail.papadakis@uni.lu, yves.letraon@uni.lu

***Abstract*—Vulnerabilities are one of the main concerns faced by practitioners when working with security critical applications. Unfortunately, developers and security teams, even experienced ones, fail to identify many of them with severe consequences. Vulnerabilities are hard to discover since they appear in various forms, caused by many different issues and their identification requires an attacker's mindset. In this paper, we aim at increasing the understanding of vulnerabilities by investigating their characteristics on two major open-source software systems, i.e., the Linux kernel and OpenSSL. In particular, we seek to analyse and build a profile for vulnerable code, which can ultimately help researchers in building automated approaches like vulnerability prediction models. Thus, we examine the location, criticality and category of vulnerable code along with its relation with software metrics. To do so, we collect more than 2,200 vulnerable files accounting for 863 vulnerabilities and compute more than 35 software metrics. Our results indicate that while 9 Common Weakness Enumeration (CWE) types of vulnerabilities are prevalent, only 3 of them are critical in OpenSSL and 2 of them in the Linux kernel. They also indicate that different types of vulnerabilities have different characteristics, i.e., metric profiles, and that vulnerabilities of the same type have different profiles in the two projects we examined. We also found that the file structure of the projects can provide useful information related to the vulnerabilities. Overall, our results demonstrate the need for making project specific approaches that focus on specific types of vulnerabilities.**

***Index Terms*—Software Security, Vulnerabilities, Common Vulnerability Exposures, Software Metrics**

I. Introduction

Vulnerabilities are a monkey on the back of developers, especially when they are developing security critical applications. Recent events with the heartbleed bug in OpenSSL demonstrate that a single vulnerability can lead to disastrous consequences by jeopardising the safety of millions of computers [1]. Such severe incidents emphasise the need for security inspection, security testing and careful code reviews. Unfortunately, vulnerabilities, unlike bugs, can (easily) remain unnoticed by both users and developers. The difficulty is that seeking for vulnerabilities requires an attacker's mindset [2] that can foresee and exploit weaknesses.

Vulnerabilities are identified based on code reviews and security testing. To perform these activities in a cost-effective manner it is mandatory to know the entities of software, e.g., modules or functions, that are likely to be vulnerable. In view of this, researchers developed classification models using static analysis to predict where inspections should be performed. These approaches do not consider specific kinds of vulnerabilities but rather aim at predicting entities that can be vulnerable with respect to any vulnerability type [3]–[5]. Also previous studied analysed a small restrictive set of the vulnerable code characteristics [3], [6]. In view of this, we investigate the metric profile of vulnerabilities according to their types. Thus, we focus on 4 properties: i) location, ii) criticality, iii) code metric profile and iv) impact on this profile when the vulnerability is fixed.

To perform our analysis we choose two security critical open-source systems with a long history of reported vulnerabilities, i.e., the Linux kernel and OpenSSL. Since vulnerabilities are an important concern for these two systems, the community behind these projects introduced special procedures for reporting them [7], [8]. This, eases our analysis since it provides a reliable way of collecting and analysing a large number of real and critical vulnerabilities.

Overall, for the needs of the present study we gather more than 2,200 vulnerable files accounting for 862 distinct vulnerabilities. We compute and report a total of 35 software metrics, which we use to built vulnerability profiles. These profiles are then linked with the location, type and criticality of the studied vulnerabilities.

In this paper, we use on the Common Weakness Enumeration (CWE) type of each vulnerability to perform our categorisation. We found that 20 different CWE types are used for Linux and OpenSSL vulnerabilities. Among those categories, 9 are the most prevalent ones and only 3 and 2 are the most critical ones in the OpenSSL and Linux kernel, respectively. We also found that most of the vulnerabilities are located on 2 and 4 directories for the OpenSSL and Linux projects, respectively. Our key findings are that vulnerability types have different (metric) profiles on the studied projects. This indicates that it is rather hard to build a generic approach operating in a cross project basis that finds all types of vulnerabilities. In particular, our results suggest that future research should focus on building a "personalised" vulnerability prediction model for every type of vulnerability. Another important finding is that vulnerabilities tend to be on specific parts of the studied projects, which, could be leveraged by the future prediction approaches.

Overall, our study makes the following contributions:

- It presents the results of a study on more than 2,200 vulnerable files, which account for 862 different vulnerability reports from two security critical systems, the Linux kernel and OpenSSL.
- It constructs a metric profile for every major type of vulnerabilities for both considered systems. This profile includes the location of the vulnerability, its criticality, its code metrics and its impact when fixing the vulnerable code.
- We found that the metric profiles differs among the different types of vulnerabilities and that the directory of the project files can provide useful information regarding the vulnerabilities.
- It introduces a new metric based on graph edit distance to compute the difficulty of fixing the vulnerable code.
- We found that the vulnerabilities that are the most complex to fix are the "Numeric Errors" in the case of Linux. In the case of OpenSSL, these are the "Information Exposure" and "Races Conditions".

The remainder of this paper is organized as follows: Sections II details the context of our study. Section III provides a motivation of our study and presents our research questions. The followed methodology is then detailed in Section IV, while our results are presented in Sections V. Section VI addresses the threats to validity of the study. Finally, Section VII concludes the paper.

II. Background

A. Vulnerability

1) Terminology: It is hard to describe vulnerabilities since they appear in various forms, e.g., they can be a consequence of bugs, insufficient security measurements or something missing. Whatever their original cause may be, their consequences are usually critical and costly. In the literature, there are several definitions for the term vulnerability, e.g., "security bug" [9] or "software weakness" [10]. In this work, we follow the definition given by the CVE initiative [11]:

"*An information security "vulnerability" is a mistake in software that can be directly used by a hacker to gain access to a system or network.*"

2) CVE, NVD and CWE: To report and collect detected vulnerabilities, there are several initiatives. In this paper, we focus on the CVE-NVD ones that are categorised according to CWE:

- The Common Vulnerability Exposures (CVE) is a referencing system for publicly disclosed vulnerabilities. It is operated by the National Institute of Standard and Technology (NIST). Each accepted and referenced vulnerability receives a unique identifier, in order to ease the sharing of data related to it. In July 2016, more than 77,700 vulnerabilities have been referenced.
- The National Vulnerability Database (NVD) is a U.S. government database for vulnerabilities referenced by the CVE system. It provides additional meta information for vulnerabilities, like the Common Vulnerability Scoring System (CVSS) score and the CWE used in this paper.
- The Common Weakness Enumeration (CWE) [12] is a community initiative to create a list of software weaknesses. This list is used by the NVD database to categorize vulnerabilities.

B. Software Under Study

In this paper, we study two major open-source software (OSS) systems, the Linux kernel and OpenSSL. Besides being open source, these two projects are widely used and involve many security-related operations. Naturally, they have lots of vulnerabilities, which have been reported over the past decade, these currently are 1460 and 159, respectively.

1) The Linux kernel: started in 1991 as a hobby by Linus Torvalds. The Linux kernel is now shipped in billions of devices (embedded in all Android devices). It is the biggest OSS with more than 19.5 million lines of code and more than 14,000 contributing developers.

While these numbers are important, they are not the main reason for choosing the Linux kernel as a candidate for our study. As mentioned earlier, the Linux kernel has to deal with many security aspects, it is the software with the second highest number of reported vulnerabilities (according to CVE). Another main reason behind this choice is the fact that the community behind the Linux kernel is well-organized. This makes it relatively easy to get relevant and reliable information on vulnerabilities. A last criterion worth mentioning is the stability of the version control system, which is used in our study to gather the vulnerable files. In the past few years, many OSS adopted git as version control system, making links to a previous version control system as reported in the vulnerability reports invalid. As the Linux kernel community created git in 2005, the Linux kernel was the first to adopt it. This gives us access to over ten years of history to study.

2) OpenSSL: Started in 1998, OpenSSL is a software library used to secure communications, or identity checking by providing an open source implementation of the SSL and TLS protocols. In 2014, two third of the web servers in the world were using OpenSSL [1].

Although OpenSSL is relatively small in size, when compared with the Linux kernel, it still has 650,000 lines of codes, 159 reported vulnerabilities and 200 contributors, recent incidents, like heartbleed [13], showed the tremendous consequences that a single vulnerability in OpenSSL can have. This makes it an interesting candidate for this study. Regarding the stability of the version control system, OpenSSL started using git in 2013. However, the transition to git was made without losing information, thus making it possible to access previous information directly from the git repository.

C. Related Work

The analysis of software vulnerabilities is an important research topic, where most of the efforts have been made in the direction of identifying vulnerabilities, i.e., automatically detecting vulnerabilities in software systems. While several approaches have been suggested, we focus here on the so-called vulnerability prediction ones, which are related to the metrics profiles we use and can be influenced by our results.

Vulnerability Prediction Modelling (VPM): Initiated by Neuhaus et al. [4], VPM is a relatively recent field of study aiming at classifying an entity as vulnerable or not, using different features extracted from the software system. This category of techniques has the advantage of being software agnostic, as they do not require specific information about it. Among the possible features to use, Shin et al., [3] [14] and Chowdhury and Zulkernine [15] [16] suggested the use of code metrics. Neuhaus et al., [4] proposed to use include and function calls of files, whereas Scandariato et al. [5] [6] advocates the use of text mining, i.e., a bag of words to build the prediction model. These approaches were replicated and compared on the Linux Kernel by Jimenez et al. [17]. The results show that vulnerability prediction models can be relatively precise with Neuhaus et al. [4] being the best predictor for future vulnerabilities. All these approaches aim at pointing out likely vulnerable parts of a system, while our study aims at profiling vulnerabilities.

The closest work to ours is the one of Bosu et al. [18], who studied the characteristics of vulnerability introducing commits, and use the same categorization we use for analysing vulnerabilities. However, our study differs in the following three points. First, there is a major difference in the used ground truths. We rely on referenced vulnerabilities from which we retrieve their patches, whereas they analyse code reviews looking for vulnerabilities. Second, our aim is to study the characteristics of real vulnerabilities, which have passed through the code review process, while they were interested in the conditions that make a commit to introduce a potential vulnerability (which was found by the code inspection). Third, we study the difference (in terms of software metrics) between the vulnerable code and its fix, while they only concern vulnerability introduction.

Other related studies that analysed vulnerabilities and their patches are due to Milenkoski et al. [19], Fonseca et al. [20] and Jimenez et al. [21]. These studies focus on specific kinds of vulnerabilities and their causes.

III. Research Questions

Due to their importance, vulnerabilities are a popular research subject. The community has organized its efforts by creating a set of software weaknesses (namely CWE) and a scoring system for criticality (namely CVSS), based on exploitability and impact metrics. An interesting starting point for our study is to identify the prevalence of the vulnerabilities according to their types and criticality. Thus, we ask:

RQ1. Which are the types of vulnerabilities that are most prevalent in Linux kernel and OpenSSL? Can we link categories with criticality levels?

The answer of this question will reveal the types were vulnerabilities are more prevalent. It will also reveal the categories with the most critical vulnerabilities.

The next step of our study regards the vulnerability location, i.e, which part of the software system a vulnerability is originated from. Together with the categorisation, this information can provide useful insights regarding the weaknesses of the different sections of the studied projects.

RQ2. Is the location of the vulnerable files linked to the vulnerability types and/or their criticality?

After studying the vulnerabilities, the next step is to analyse vulnerable files, i.e., files that needed to be modified to fix the vulnerability. This level of granularity for analysis is generally used by VPM as it was considered as actionable by Microsoft developers [22]. Thus, the result of this analysis could help VPMs by assessing the prediction power of some characteristics of vulnerable files. This leads to the following question:

RQ3. What are the characteristics of vulnerable files per considered type?

Another interesting point that has so far not received much of attention by the literature is the extent to which a vulnerability fix impacts a file. For example, are those fixes complex or simple? Can we observe different patterns when fixing same specific types of vulnerabilities? All these concerns guide us to our next research question:

RQ4. What is the impact of fixing a vulnerability on the metric profile we use?

An analysis of the impact could help to flag some commits as vulnerability fixes, as well as provide some insights on the vulnerability fixes.

TABLE I
Vulnerability Dataset Statistics

	Linux kernel	OpenSSL
Num of Vulnerabilities CVE	768	95
Num of Commits	899	382
Num of vulnerability types (CWE)	20	11
Vulnerable Files/ Unique	1615/951	619/102

IV. Methodology

A. Dataset

Our dataset contains all vulnerabilities from the two studied systems reported in the NVD database for which it is possible to retrieve patches. Since NVD does not store any patches, we mine them from the version control systems by identifying the vulnerability fixing commits. To retrieve these commits we use three different strategies:

- Extracting commit hashes directly from CVE references
- Gathering commits that mention a CVE number
- Retrieving all git commits mentioning in their message a bug ID that is also mentioned in one of the CVE references

We then retrieve all the files from these commits in their vulnerable and patch state.

Table I presents the statistics of our dataset. Overall, we managed to collect data representing a percentage of 52% of the possible CVE identifiers for the Linux kernel, and 59% for the OpenSSL. The table records the number of commits, the number of different vulnerability types (CWE) and the

number of vulnerable files that were retrieved. Note that a vulnerability can be fixed in more than one commit and a file can be vulnerable several times.

B. Characterization of Vulnerable Files and Fixes

We considered 35 metrics to profile the vulnerable files. These are categorised as follows:

- Basic Metrics: lines of code, blank lines, commenting lines, comment density, preprocessor lines, number of variable, number of declared functions. These metrics provide information regarding the profile of the vulnerable files.
- Code Metrics: all variants of cyclomatic complexity (strict, modified and standard), essential complexity, maximum nesting, fan in and fan out. Note that these are function-level metrics and we are working at the file level so we computed for each file the metrics on all the function and kept the maximum, average and sum values. These metrics characterise the structure of the code and their definition can be found at [23].
- Code churn: number of changes and number of lines added, deleted, modified in the history of the file.
- Developer history: number of developers currently working on the file (git blame), number of developers that have worked on the file.

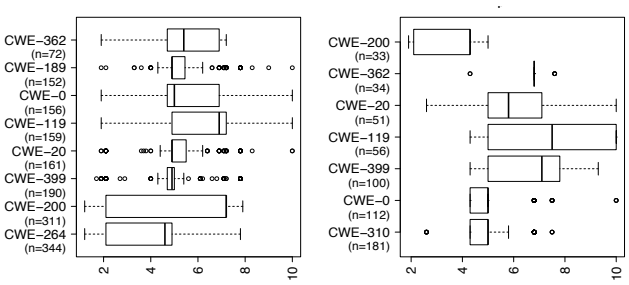
Regarding the impact of a vulnerability fix we measured the delta of all the previously suggested metrics between the vulnerable file and the fixed one. This provides information related to the nature of the patch that fixes the vulnerability. However, it does not give any general overview on the impact of the change on the code. In view of this, we introduce a new metric detailed in the following subsection.

C. Graph Edit Distance (GED)

Measuring the impact of a vulnerability fix is not an easy task. Making a 'diff' on the fixed and vulnerable files, which is a common practice, only reveal changes at the line level and does not consider the control flow. To measure the impact on the program flow, we compute the control flow graph (CFG) of the related functions before and after the modifications. Then, we compute a graph edit distance (GED) between the two graphs. Graph edit distance was first formalised by Sanfeliu et al., [24] and is used to measure the similarity between two graphs. A survey on its use has been conducted by Gao et al., [25]. The main idea here is to evaluate the minimum edit "cost" of going from the vulnerable CFG to the fixed CFG, by attributing different costs for an edge or a vertex of the graph replacement, deletion or insertion. For our study, we chose to attribute a value of 2 for replacement, 1 for insertion, 1 for deletion. This measure will indicate to what extent the patch modified the control flow of the function. As this measure is also a function level metric, we compute the GED for all functions of a file and sum them up after. The GED of an unmodified function being 0. Our implementation of this GED is based on a github project available at [26].

TABLE II
DEFINITION OF CWE RETAINS FOR THIS STUDY

CWE	Definition
20	Improper Input Validation
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
189	Numeric Errors
200	Information Exposure
264	Permissions, Privileges, and Access Controls
310	Cryptographic Issues
362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
399	Resource Management Errors
0	Vulnerabilities without a CWE Number



(a) Linux Kernel (b) OpenSSL

Fig. 1. Criticality (CVSS score) per type of vulnerability (CWE), n indicates the number of vulnerable files in the dataset for this category

D. Experimental Process

Once all vulnerable files are gathered, i.e., by following the procedure described in Section IV-A, we compute our metrics both in the state before and after the vulnerability fixes. We then compute all deltas and the graph edit distances. All of our experiments, were produced using the the HPC facilities of the University of Luxembourg [27]. Once in possession of all the metrics we analysed our results by grouping the vulnerable files according to their CWE types.

V. RESULT

A. RQ.1: Most reported types of vulnerabilities and criticality

When analysing our data, we found that specific types of vulnerabilities are scarce. As our goal is to identify trends, we filter them out and focus only on the most common ones. Thus, we choose a threshold of 50 vulnerabilities for the Linux Kernel and 30 for OpenSSL , i.e., each type of vulnerability to be present in at least 50 or 30 vulnerable files. This left us with the 9 types of vulnerabilities that are more prevalent in the studied systems. These 9 vulnerability types are reported in Table II.

After identifying the vulnerability types, we focus on their criticality. Figure 1 presents these results, ordered (from top to bottom) by the number of vulnerable files present in each category. A first observation that we can make is that the most represented categories (order) of one system are

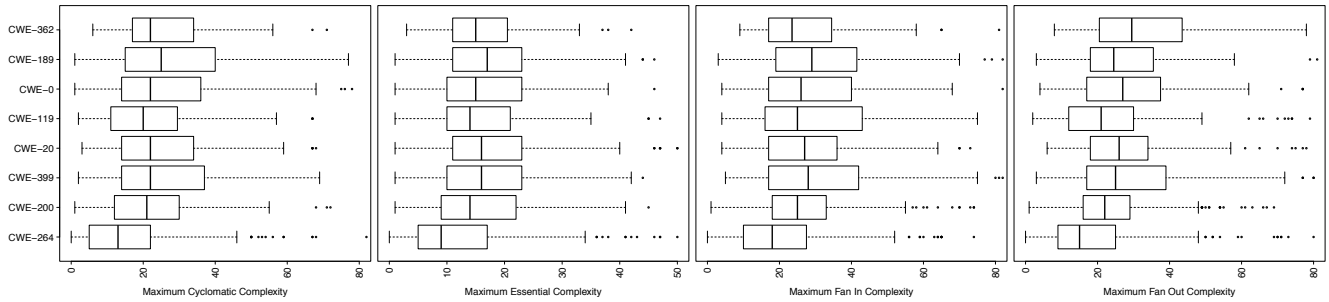


Fig. 2. Maximum complexity of Linux Kernel vulnerable files

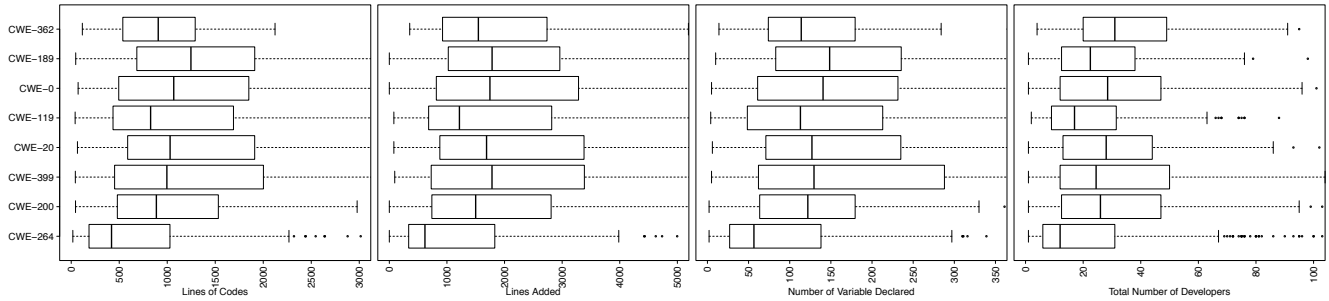


Fig. 3. Metrics of Vulnerable Files for the Linux Kernel

TABLE III
LOCATION OF THE VULNERABLE FILES IN OPENSSL

Directory	Number of Vulnerable Files	CVSS Score	most represented CWE (number of vulnerable files)
apps	9	6.77	CWE-20 /CWE-399 (4)
crypto	286	5.82	CWE-310 (90)
engines	9	6.67	CWE-399 (6)
ssl	314	5.52	CWE-310 (90)

TABLE IV
LOCATION OF THE VULNERABLE FILES IN THE LINUX KERNEL

Directory	Number of Vulnerable Files	CVSS Score	most represented CWE (number of vulnerable files)
arch	241	4.46	CWE-264 (120)
crypto	81	2.35	CWE-264 (66)
drivers	239	4.95	CWE-119 (55)
fs	339	5.60	CWE-200 (95)
kernel	117	5.92	CWE-200 (49)
mm	71	5.83	CWE-264 (22)
net	423	5.26	CWE-200 (76)
security	38	5.85	CWE-119 (12)

not the ones of the other. This is due to the functionality differences of the studied systems. Another observation is that the most critical type of vulnerabilities for the Linux kernel is the "Information Exposure" (CWE-200), whereas its the less critical for OpenSSL. With respect to CVSS score the most consistent category over the two software systems is the "Improper Restrictions Of operations within the bounds of a memory buffer" (CWE-119) with a relatively high average criticality of 7 in both cases. This is even the most critical one for OpenSSL. This buffer problem is extremely risky as it is related to memory and can impact other programs.

Overall, our results show that out of the 20 types of vulnerabilities, 9 are prevalent and among them the most critical ones (with CVSS score above 6) are the CWE-200 and CWE-119, for Linux and CWE-119, CWE-399 and CWE-362, for the OpenSSL.

B. RQ. 2 Location of the vulnerabilities

Table III and IV present the results related to the actual location of vulnerable files within the project file structure. It is important to note that only the main directory appears here. This is to keep consistency between OpenSSL and Linux due to the fact that OpenSSL is not using subdirectories. One interesting result is that in OpenSSL the vulnerabilities are either emanating from the crypto directory or the SSL one. This may be explained by a previous observation that the most represented category is related to the "Cryptographic" issues (Fig. 1(b)). Interestingly, the most critical vulnerabilities are located on the "apps" directory, but it involves much lower number of vulnerabilities. Overall, among the two most vulnerable directories (crypto and SSL), those in 'crypto" directory are more critical.

In the Linux kernel, the directory with the most vulnerable files is the "net" directory, which is in charge of the network functionalities and has "Information Exposure" vulnerabilities. Interestingly, the problem of permission and privileges seems to occur mostly in the "arch" and "crypto" directories, whereas

buffer errors are mostly present in the "drivers" and "security" directories. Looking at the criticality, the "kernel" directory seems to be the more at risk followed by the "mm" responsible for the memory management one.

Overall, our results suggest that the file structure of the projects, filenames and categories can provide useful information related to the nature of the vulnerabilities. As we will see in the following sections, this is complementary to the information provided by software metrics.

C. RQ. 3 Characteristics of vulnerable files

Due to space limitations we present the 8 software metrics that have the strongest discrimination power. The complete set of our results can be be found at: http://www.jimenez.lu/Research/VulnerabilityAnalysis/vuln.html

1) Linux Kernel: Figures 2 and 3 presents the results related to the selected metrics. A first observation indicates that vulnerable files related to "Permissions, Privileges, and Access Control" (CWE-264) contain less complex functions than any other type of vulnerabilities. They also have fewer lines of code, lines added, variable declared and a smaller group of developers working on it. This indicates that files dealing with permission rights do not require complex algorithms and are less likely to be modified once been written.

Files related to "Numeric Errors" (CWE-189) appear in the the most complex functions (according to "Cyclomatic Complexity", "Essential Complexity" and "Fan In"). The same files are also having the highest number of lines of codes and declared variables. This may suggest that numeric error, i.e., improper calculation or conversion of numbers, are more likely to be found in complex functions than in simple ones.

High "Fan Out" values, i.e., the number of called functions plus global variables, and high number of developers result in "Race Conditions" problems. This indicate that code parts related to concurrency have also high "Fan Out", which in term requires special attention. These files are also important and seem to have a central interest in the project as there is also a high number of developers working on them.

We also observe that vulnerable files without a category (CWE-0) have average values for all metrics, which may suggest that these vulnerabilities are a mixture of categories.

2) OpenSSL: Figures 4 and 5 present the results for OpenSSL. A first point is that there is a difference between the maximum cyclomatic and essential complexity of the OpenSSL than of the Linux kernel, while interestingly the number of lines of code stay in the same range of values.

The category of vulnerable files with the higher score are those that are uncategorized vulnerabilities (CWE-0). This is also true for the number of lines of code and declared variables. These might indicate that uncategorized vulnerabilities are of other categories than the existing ones.

Considering "Race Conditions" (CWE-362) vulnerable files, we observe that they are due to the involvement of many developers (like in the Linux kernel). On the side of "Cryptographic Issues" (CWE-310) that is the most represented type of vulnerabilities in OpenSSL, we observe a low maximum complexity compared to other types of vulnerabilities, except for "Fan Out" result.

D. RQ. 4 Impact of vulnerability fixes

Here we compute the delta of the employed metrics (between the vulnerable and fixed version of the systems) and the graph edit distance.

1) Linux Kernel: Figure 6 presents the results for the selected metrics. Regarding the complexity, the impact of a fix is similar for most of vulnerabilities except for "Information Exposure", ones for which there is a lower complexity. This result is surprising, as one could expect an increase of complexity from additional checks. Looking at graph edit distance metrics, fixes for CWE-264 are the ones that are less impacted whereas the fixes for " Numeric Errors" are the ones with the greatest impact. This was expected as those vulnerabilities were the ones with the higher complexity observed for most metrics.

2) OpenSSL: Figure 7 shows the OpenSSL results. A first observation is that there is much larger variation from one category to another, than in the case of the Linux kernel. Fixes of CWE-119 files seem to have a larger impact on the cylomatic complexity increasing it by a value of two, on average, and the highest delta in lines of codes. In the "Fan In", i.e., the number of function calls plus global variable reads, the highest variation is observed for CWE-362 fixes. Regarding graph edit distance, the fixes with the most impact on the CFG are the ones from "Information Exposure" (CWE-200) files followed by "Races Conditions" CWE-362. Interestingly, these two types are also among the top ones of the Linux kernel for this metric.

VI. Threats to validity

A. Construct Validity

The datasets used for this study are automatically generated using git commit messages and the NVD database. Thus, imprecise information in the NVD or misleading commit messages could generate noise in our data. Similarly, the categorization of vulnerabilities might be inconsistent, since this is a manual process performed by different people. Thus, it is likely that different points of view on which category a vulnerability belongs to might exist. However, given the well-organized community behind these projects with strict guidelines [28], we believe that this could only be the case for a small percentage of the vulnerabilities.

In this study, we consider as vulnerable files, every file that had to be modified to fix a vulnerability. Thus, all the files from a fixing commit were added in the dataset. Yet, some commits might include fixes for some other things than this vulnerability, hence adding files unrelated to the vulnerability to our dataset. As this is against common practices, it should not impact our results. Moreover, our aim is to profile vulnerable files so that developers can focus on such files.We believe that the modified code parts are indeed relevant to the vulnerable code since they are fixing vulnerabilities.

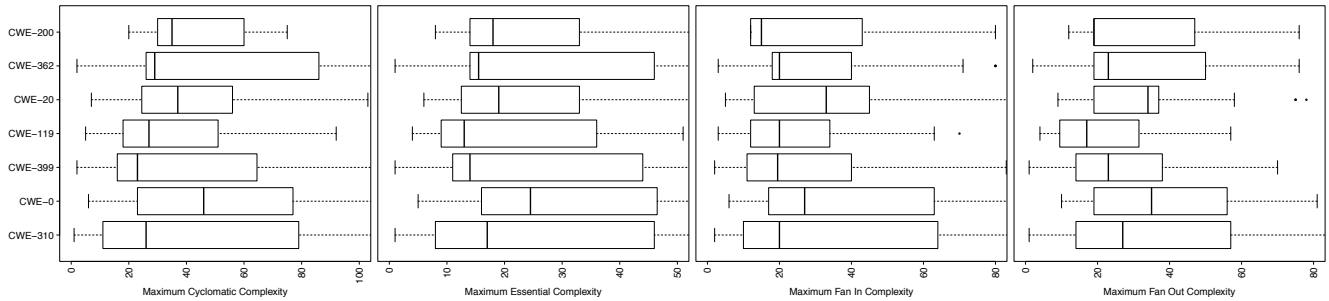


Fig. 4. Maximum complexity of OpenSSL vulnerable files

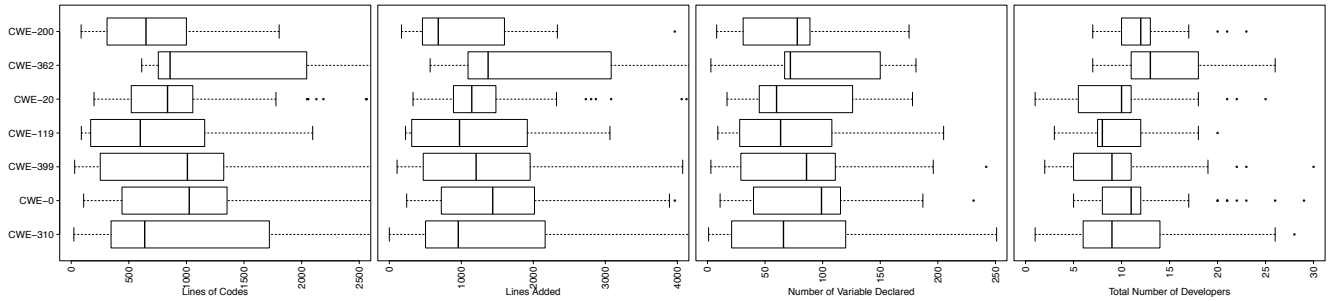


Fig. 5. Metrics of Vulnerable Files for OpenSSL

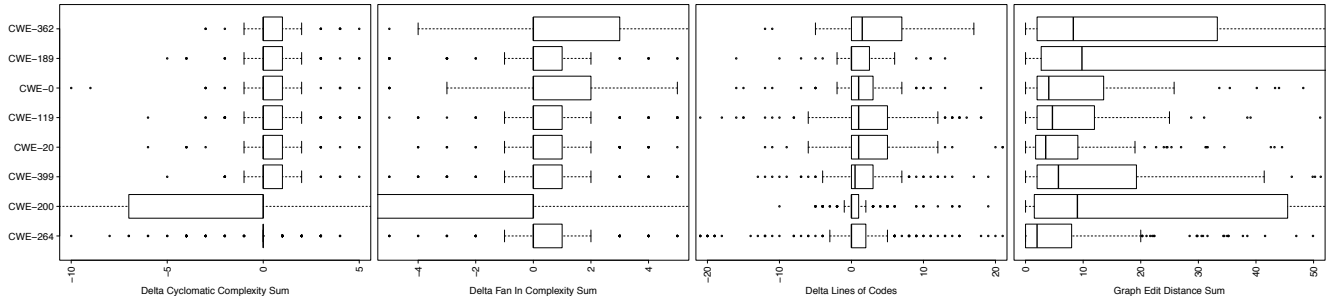


Fig. 6. Impact of Fixes for Linux kernel

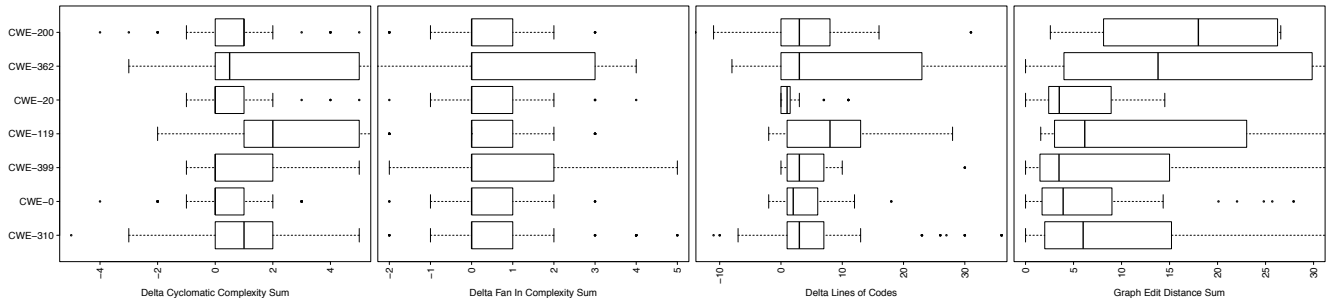


Fig. 7. Impact of Fixes for OpenSSL

B. Internal Validity

Potential bugs in our implementation may also influence our results by providing incorrect measures. To reduce these threats, we carefully tested and verified our implementation. Moreover, as metrics for all files are computed the same way, this should not have much influence on particular profiles we report as similar variations, in the measurements, should be observed with other tools.

C. External Validity

This study is limited to two open source software systems written in C. Thus, our results might not be generalizable to systems written in other languages. This is partly indicated by our results which show that different profiles exist, between the two projects, for the same types. However, these two projects are typical examples of safety critical applications, they are well organized and have a long history of vulnerabilities that includes a rather large number of vulnerabilities.

VII. Conclusion

We analysed the characteristics of the Linux kernel and OpenSSL vulnerabilities. We found that different types of vulnerabilities have different profiles. On the one hand, this finding indicates that there is no "pot of gold" for vulnerability prediction methods, i.e., approach that finds all types of vulnerabilities, while on the other it suggests that current approaches could be tuned to target specific types of vulnerabilities.

Overall, the analysis of 2,200 vulnerable files, related with 862 vulnerabilities reveal that they are of 20 types. However, 9 of them are the most prevalent ones and only few are critical (3 for OpenSSL and 2 for the Linux kernel). These results suggest that future research should focus on building specialised models targeting these critical types of vulnerabilities.

Another important finding is that the identified profiles are system specific. This means that it is hard to draw conclusions for all types of vulnerabilities since vulnerabilities have different profiles in the two systems we study. This suggests that the use of prediction models that are trained on one project may fail on another and hence the creation of cross-project vulnerability models, as attempted by existing approaches [3], [5], seems to be quite hard if not impossible.

Additionally, our results show that the location of the vulnerable files can provide useful information on where and which type of vulnerabilities to look for. Also our results suggest that vulnerability criticality is related to its location. Interestingly, file location is not considered by any of the existing vulnerability prediction methods. A possible reason for this is that location is a nominal property linked to the software system under analysis and can only be used for specific project prediction and not for cross system ones.

Finally, our results indicate that fixing vulnerabilities in Linux does not involve many changes (from source code editing point of view). In OpenSSL the fix process is more complex than in Linux and require many changes, located in different parts of the code.

In future work we plan to extend our analysis on additional safety critical systems and investigate whether "personalized" prediction models, i.e., project-specific or type-specific, can be developed. We also plan to assess the practical benefits of such models with respect to existing ones.

References

[1] Bug in openssl opens two-thirds of the web to eavesdropping. [Online]. Available: http://arstechnica.com/security/2014/04/critical-crypto-bug-in-openssl-opens-two-thirds-of-the-web-to-eavesdropping/

[2] G. McGraw and B. Potter, "Software security testing," *IEEE Security & Privacy*, vol. 2, 2004.

[3] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE TSE*, vol. 37, Nov. 2011.

[4] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *CCS'07*.

[5] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," *IEEE TSE*, vol. 40, Oct. 2014.

[6] J. Walden, J. Stuckman, and R. Scandariato, "Predicting Vulnerable Components: Software Metrics vs Text Mining," in *ISSRE'14*.

[7] Linux procedure for security bugs report. [Online]. Available: https://www.kernel.org/doc/Documentation/SecurityBugs

[8] Openssl procedure for security bugs report. [Online]. Available: https://www.openssl.org/news/vulnerabilities.html

[9] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, 2014.

[10] Ics/scada top 10 most dangerous software weaknesses. [Online]. Available: http://www.toolswatch.org/wp-content/uploads/2015/11/ICSSCADA-Top-10-Most-Dangerous-Software-Weaknesses.pdf

[11] Definition of vulnerability. [Online]. Available: https://cve.mitre.org/about/terminology.html

[12] Cwe home page. [Online]. Available: https://cwe.mitre.org/data/

[13] Heartbleed home page. [Online]. Available: http://heartbleed.com

[14] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, pp. 25–59, Feb. 2013.

[15] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *SAC'10*.

[16] "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, 2011.

[17] M. Jimenez, M. Papadakis, and Y. L. Traon, "Vulnerability prediction models: A case study on the linux kernel," in *SCAM'16*, 2016.

[18] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: an empirical study," in *FSE'14*, 2014.

[19] A. Milenkoski, B. Payne, N. Antunes, M. Vieira, and S. Kounev, "Experience report: An analysis of hypercall handler vulnerabilities," in *ISSRE'14*.

[20] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira, "Analysis of field data on web security vulnerabilities," *Dependable and Secure Computing, IEEE Transactions on*, vol. 11, no. 2, March 2014.

[21] M. Jimenez, M. Papadakis, T. F. Bissyande, and J. Klein, "Profiling android vulnerabilities," in *QRS'16*. IEEE, 2016.

[22] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *HotSoS'15*.

[23] Scitool complexity metric. [Online]. Available: https://scitools.com/support/metrics_list/?metricGroup=complex

[24] A. Sanfeliu and K. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 13, 1983.

[25] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and Applications*, vol. 13, 2010.

[26] Graph edit distance implementation. [Online]. Available: https://github.com/haakondr/NLP-Graphs/

[27] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *HPCS'14*.

[28] Linux coding style. [Online]. Available: https://www.kernel.org/doc/Documentation/CodingStyle