A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned

Bingchang Liu[*]
liubingchang@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Guozhu Meng[*][†]
mengguozhu@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Wei Zou[*]
zouwei@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Qi Gong
gongqi@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Feng Li[*]
lifeng@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Min Lin
mingo.limz@gmail.com
Institute for Network Science and
Cyberspace, Tsinghua University

Dandan Sun
sundandan@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Wei Huo[*]
huowei@iie.ac.cn
Institute of Information Engineering,
Chinese Academy of Sciences

Chao Zhang[‡]
chaoz@tsinghua.edu.cn
Institute for Network Science and
Cyberspace, Tsinghua University

ABSTRACT

The number of vulnerabilities increases rapidly in recent years, due to advances in vulnerability discovery solutions. It enables a thorough analysis on the vulnerability distribution and provides support for correlation analysis and prediction of vulnerabilities. Previous research either focuses on analyzing bugs rather than vulnerabilities, or only studies general vulnerability distribution among projects rather than the distribution within each project. In this paper, we collected a large vulnerability dataset, consisting of all known vulnerabilities associated with five representative open source projects, by utilizing automated crawlers and spending months of manual efforts. We then analyzed the vulnerability distribution within each project over four dimensions, including files, functions, vulnerability types and responsible developers. Based on the results analysis, we presented 12 practical insights on the distribution of vulnerabilities. Finally, we applied such insights on several vulnerability discovery solutions (including static analysis and dynamic fuzzing), and helped them find 10 zero-day vulnerabilities in target projects, showing that our insights are useful.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **General and reference → Empirical studies**.

[*] Also with: School of Cyber Security, University of Chinese Academy of Sciences.
[†] Corresponding Author
[‡] Also with: Beijing National Research Center for Information Science and Technology

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380923

KEYWORDS

Empirical Study, Vulnerability Distribution

ACM Reference Format:
Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380923

1 INTRODUCTION

Software vulnerability is one of the major threats to cyber security. For example, vulnerabilities like HeartBleed [57], SandWorm [58] and DirtyCow [56] have posed great risks on millions of users. Due to recent advances of modern vulnerability hunters (e.g. AFL [8], AFLGo [12], Steelix [33], CollAFL [19]), vulnerabilities are being reported at a striking speed. The number of exposed vulnerabilities recorded by National Vulnerability Database (NVD) [40] already exceeds forty thousands, let alone the vulnerabilities which remain unknown to the public. This large volume enables a thorough analysis of vulnerability distribution, and provides support for correlation analysis of vulnerabilities, which in turn helps finding new vulnerabilities.

For instance, as shown in Figure 1, we found 5 new vulnerabilities (marked as `NVx`) when studying some existing vulnerabilities (marked as `MSxx`) in `Microsoft Windows Journal`. We figure out that, all these new vulnerabilities are close to existing vulnerabilities (marked as `MSxx`) in the call graph. Some of them even reside in the same functions as existing vulnerabilities, or have similar code snippets. It therefore implies that, vulnerabilities in a project could have locality in space (e.g., function locations) and patterns (e.g., code structures), and such locality could be utilized to locate new vulnerabilities around known vulnerabilities in the project. Motivated by this example, we intend to analyze the vulnerability

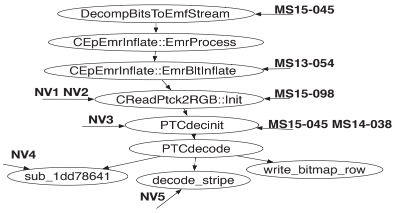


Figure 1: A real-world example: vulnerabilities have a strong correlation which could help finding new vulnerabilities.

distribution within projects, in order to investigate if such vulnerability locality is common in individual projects. If so, we further intend to investigate whether such locality could be utilized to find new vulnerabilities around known vulnerabilities.

Prior works [10, 14, 17, 24, 25, 52] have studied a large number of bugs and provided insights for better understanding the distribution and root causes of bugs. These insights are further utilized to detect more related bugs [9, 53, 54]. However, these approaches and insights of bugs cannot be well shifted to vulnerabilities, since vulnerabilities exhibit many different characteristics from bugs. For instance, vulnerabilities are much more rare than bugs. Another line of works [15, 29, 39, 51] concentrate on utilizing *individual* features of vulnerabilities, *e.g.*, function calls and code complexity, to predict new vulnerabilities. Some other works [18, 26, 41, 47, 62] focus on studying the life-cycle of vulnerabilities and how to fix them. The work [32] studied the vulnerability distribution across projects. However, existing works did not study the vulnerability distribution within projects, and thus are not able to analyze vulnerability locality in individual projects.

To study vulnerability distribution within projects, there are several challenges to address. First, there are no qualified datasets available for a large scale analysis. Existing vulnerability datasets focus on covering diversified projects, but not all vulnerabilities for a given set of projects. Second, a large scale data analysis is time-consuming and in general challenging.

In this work, we make a first attempt to study the distribution of and correlation between vulnerabilities within projects on a large scale. We select 5 typical open-source projects `Linux-kernel`, `FFmpeg, ImageMagick, OpenSSL and php-src` as the analysis objects. For each project, we collected all its associated vulnerabilities by aggregating multiple data sources, including NVD entries, BugZilla reports, security bulletins and GitHub commits. We then attempt to answer the following questions:

RQ1: **Vulnerability Distribution.** How are vulnerabilities distributed? Can it be used for hunting new vulnerabilities?

RQ2: **Vulnerability Dependency.** Why are vulnerabilities oftentimes found within a short code range? Is there any dependency between vulnerabilities?

RQ3: **Vulnerability Recurrence.** Why do vulnerabilities repeatedly occur in the same function? Is it a general case?

To this end, we analyze the collected vulnerability dataset as follows. We first associate each vulnerability with its responsible code commits, then localize the vulnerable code snippets related to each vulnerability, and then build the call-graph for the vulnerable code snippets. Further, based on the extracted information, we analyze the vulnerability distribution over space, type, and responsible developers, as well as the dependency between vulnerabilities in the call graph and the vulnerability recurrence phenomena.

We find that, the vulnerability distribution follows the Pareto law [45] on the dimension of space (file, not function), type and responsible developers. Moreover, most (more than 60%) vulnerable functions have at least one vulnerable neighbor function located in a 2-jump range in the call-graph. In other words, most vulnerable functions do not exist alone. We also find that, incomplete or wrong patches for vulnerabilities are the most common reasons for the existence of high-frequency vulnerable functions, whereas the patching workflow ROPO (Report One issue and Patch One position based on the reference PoC) is another important reason.

In summary, we make the following contributions.

- *Large-scale vulnerability dataset.* Aiming at studying the vulnerability distribution within projects, we spent 5 person months efforts to build a largest-to-date vulnerability dataset regarding a set of open source projects. We share this dataset to the public [1], to facilitate other researches.
- *Thorough empirical study.* Different from the prior work, our study unveils the characteristics of vulnerability distribution over several dimensions including file, function, type and responsible developers. Moreover, we also investigate the correlation of vulnerabilities so as to uncover the causes of their presence.
- *Practical insights.* We present several practical insights based on the analysis results. In particular, we find that most vulnerabilities do not exist alone. We also find that, imperfect vulnerability patches and the patching workflow ROPO are general phenomena, causing vulnerabilities repeatedly occur in some functions. We also propose a solution to mitigate the ROPO issue.
- *New discoveries.* Based on the concluded insights, we propose a solution to use them to guide vulnerability discovery, and have applied them to several vulnerability discovery solutions including static analysis and dynamic fuzzing, and help them successfully find 10 zero-day vulnerabilities in target projects.

2 APPROACH

Figure 2 presents the overview of the approach we used in this empirical study. First, we select several popular open source projects that represent varying functionalities. Then, we aggregate multiple vulnerability data sources, including NVD/CVE and BugZilla, to collect all description information of vulnerabilities associated with target projects. We also crawl all code commits of target projects from GitHub, to support detailed vulnerability analysis.

Further, we analyze the relationship between NVD/CVE vulnerability entries and GitHub commits, to recognize the commits related to each vulnerability. Then, we localize the vulnerable code snippets and construct their call graphs. At last, we perform multiple analyses on these vulnerabilities and conclude many take-home messages for future research. Moreover, we develop proof of concepts based on these findings to evaluate their practicality.

2.1 Target Selection

We select 5 popular open-source projects, written in C/C++ language, as our analysis object: FFmpeg [2], ImageMagick [3], OpenSSL [6], PHP-SRC [7] and Linux kernel [5]. On one hand, these

[1] https://github.com/twelveand0/CarrotsBlender

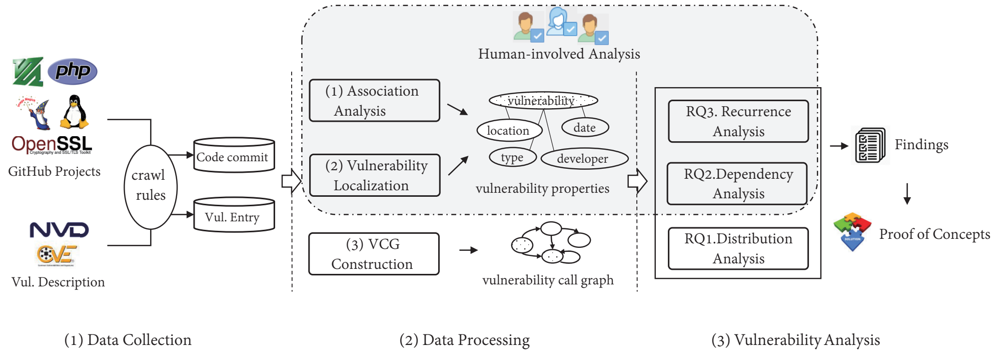


Figure 2: The overview of the empirical study

Table 1: Statistics of our dataset, including the number of stars, forks and commits of each project obtained from GitHub, and the number of associated CVE entries in NVD, as well as the number of vulnerabilities in our dataset, including those with CVE IDs and without CVE IDs.

Project	# Stars/Forks	# Commits	# CVE	# Vulnerabilities in our dataset		
				with CVE	w/o CVE	total
FFmpeg	16,266/5,911	93,265	289	256	652	908
ImageMagick	3,083/543	15,202	458	280	476	756
OpenSSL	11,099/4,807	23,444	192	130	2	132
PHP-SRC	24,471/5,622	111,821	576	346	80	426
Linux	79,149/27,578	841,891	2291	1716	0	1,716
Total		**1,085,623**	**3,806**	**2,476**	**1,462**	**3,938**

projects span a wide range of functionalities. FFmpeg is a multimedia framework, ImageMagick is a representative raster/vector image file processing software suite, OpenSSL is an implementation of secure communication protocols, PHP-SRC is the official interpreter for PHP language, and Linux kernel is one operating system. The diversity intuitively involves developers of different fields and varying programming styles. Therefore, more types of vulnerabilities can be covered. On the other hand, every project has at least hundreds of CVE entries in NVD [40], as shown in Table 1. Notorious vulnerabilities like ImageTragick [4], DirtyCow [56] and HeartBleed [57] are all discovered in these projects.

2.2 Data Collection

To obtain sufficient and comprehensive information for vulnerabilities, we collect code commits from Github and vulnerability entires from NVD/CVE for each project.

Code commits. We scrape all commits from the selected Github projects with an implemented web crawler in 8,000+ LOC of Python. The crawler leverages GitHub developer APIs [20] and downloads changed code, issues, pull requests, and responsible developers for each commit. To reduce the data volume, we set up a crawl rule that filters out the commits without making any semantic-aware change. More specifically, the changes to configuration files, code comments and spacing are not substantially related to vulnerabilities, and thereby not embodied in our dataset. After applying this rule, we get a total of 1,085,623 commits as shown in Table 1.

Vulnerability entries. It is not always about security when a developer makes a commit to the project. Functionality update, performance optimization, and software refactoring are all possible reasons. As such, we have to first identify vulnerability-related commits before the evaluation. To this end, we resort to NVD [40] and CVE [34] which maintain a large corpus of vulnerabilities, and recognize the real vulnerability-related commits. Not all CVE entries are useful in this task, so we restrict the entries by specifying the vendors and products. For example, FFmpeg vulnerabilities can be distilled by searching the entries with the product "FFmpeg" and vendor "FFmpeg". In this way, 3,806 CVE entries are obtained.

2.3 Human-involved Analysis

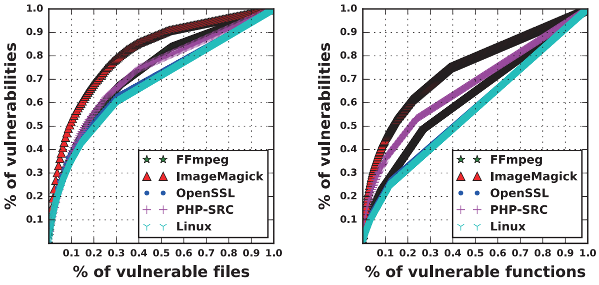
There are several tasks requiring professionals' involvement as indicated in Figure 2. In this study, we recruited 3 security experts, all of whom have more than 5-year experience in vulnerability research. In total, they have found dozens of unknown vulnerabilities in both close/open-source software, and more than 30 vulnerabilities have been assigned with CVE identifiers. One of them was even listed in the 2016 MSRC Top 100 Security Researchers.

Cooperation. To ensure the correctness of the manual analysis results, we employ a customized peer review process. At first, each task is analyzed by two experts. The task finished if they could reach an agreement. Otherwise, the third expert is involved to resolve the divergence. If she/he agrees with either one of the previous two experts, then we follow the simple majority rule to yield the final analysis result. In the worst case, three experts report three different results, then we follow a conservative principle to resolve the issue. More specifically, three experts will discuss together and yield a conservative result. Take the CWE labelling task as an example, given a CVE vulnerability, if three experts report three different CWE labels, then a generalized CWE label which covers them or an 'UNKNOWN' label will be yielded as the final result.

2.4 Data Processing

As Figure 2 shows, we process the collected data in three steps: *association analysis*, *vulnerability localization* and *VCG (Vulnerability Call Graph) construction*. They are detailed as follows.

2.4.1 Association Analysis. We first try to associate CVE entries with responsible commits. If one commit is found contributing to a CVE entry, we regard it as a vulnerability-related commit. Note that, CVE entries in the NVD/CVE database often have references to external links, including links of responsible commits in the



(a) Files.

(b) Functions.

Figure 3: Alberg diagrams showing the percentage of vulnerabilities over the percentage of vulnerable files/functions for each project. The x axis depicts the percentages of vulnerable files/functions sorted by the number of vulnerabilities contained in descending order. The y axis is the cumulative percentages of vulnerabilities in these files/functions.

vulnerable project's GitHub repository. With this guidance, we implement a tool to parse these references and extract related GitHub commit links, and automatically build the connections to commits for most CVE entries. For the remaining CVE entries, we spend three person months to manually investigate their corresponding commits, by reviewing the detailed vulnerability descriptions from multiple sources, including reports from Google OSS-Fuzz, RedHat BugZilla and a wide range of security analysis blogs.

Data quality. To make the vulnerability dataset error-free and precise, we exert the following efforts: 1) *Redundancy removal.* If one vulnerability is associated with multiple commits, we check whether the commits are of the same content but appear in different branches. If so, we just retain the newest commit. 2) *Type correction.* Each vulnerability in the dataset is classified into a CWE category. However, existing classification results are oftentimes wrong or imprecise (cf. Section 3.2). We therefore manually correct the category of each vulnerability, following the CWE-1000 classification criteria. Take vulnerability CVE-2016-2549 in project Linux-kernel as an example, it is labeled with CWE-20 (Improper Input Validation) and associated with the commit `2ba1fe7` in NVD. However, after reviewing the CVE description, RedHat BugZilla comments, messages of the responsible commit and the corresponding patch, we determined that it is actually a deadlock problem, and reach a consensus of labeling it as CWE-833 (Deadlock).

As depicted in Table 1, there are in total 3,806 vulnerabilities recorded in the CVE/NVD database for the selected projects. After the automated and manual association analysis, 2,476 CVE entries are successfully associated with their responsible commits. Note that, during manual analysis, we also find a large number of commits that are created for fixing security bugs not listed in CVE/NVD. We totally find 1,462 such commits, and associate them with unlisted vulnerabilities, and put into our dataset as well.

In total, we obtain a cross-verified vulnerability database, including 3,938 vulnerability entries. Each entry has been associated with responsible commits and related metadata information, including locations, the commit date and responsible developers of vulnerabilities.

2.4.2 Vulnerability Localization. After identifying the vulnerability-related commits, we seek to localize vulnerable code in a fine-grained granularity. By virtue of the unified diff format [27], the changes in commit mainly occur in two programming constructs–*function*, and *data structure*. For a change falling into a function, we need to recover previous vulnerable code. This is fulfilled with the built-in command "`git reset`", resetting the current HEAD to its parent commit. Additionally, we handle the following exceptions during analysis: 1) If a commit does not delete any lines and only has additions, we first identify the key vulnerability variable(s) influenced by the added code chunk and then determine the locations of variable definition or reference as vulnerable points. 2) If a commit adds a new function, this function is not treated as vulnerable.

2.4.3 VCG Construction. We aim to explore whether these vulnerabilities have semantic relations in between. Here we use vulnerability call graph to represent these relations.

Definition 1. *A vulnerability call graph is a tuple (F, V, δ) such that F is a non-empty finite set of functions, V is a non-empty finite sets of vulnerable functions where $V \subseteq F$, and $\delta \subseteq F \times F$ is the function call.*

We conduct a lightweight static analysis on top of Doxygen [55] to construct the call graph of these vulnerable functions. This tool is context insensitive and cannot handle dynamic dispatch, a trade-off between computation costs and gains. In the course of graph construction, we address two challenging problems: *same function declarations in different files* by assigning a unique identifier to each function, and *polymorphic functions* by taking into account fully qualified function names. It is worth mentioning that since some vulnerable functions are removed from the branch over time, they would not be present in the constructed call graph.

3 RQ1. VULNERABILITY DISTRIBUTION

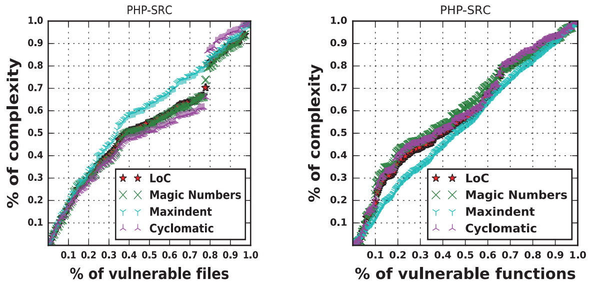
In this section, we conduct an analysis to reveal the vulnerability distribution in multiple dimensions, particularly spatial distribution, type distribution, and developer distribution.

3.1 Spatial Distribution

The study of vulnerability's spatio-temporal distribution can facilitate the prediction of where and when vulnerabilities are prone to occur. In this section, we take into account the collected vulnerabilities as well as their locations and occurring dates, and tease apart the characteristics, respectively.

The spatial characteristics are measured in two granularity levels: file and function. We sort the vulnerable files in terms of the number of contained vulnerabilities in descending order, and compute the cumulative probabilities for vulnerabilities as shown in Figure 3a. For ImageMagick, the top 30% of vulnerable files account for about 80% of vulnerabilities, while for Linux, 60% of vulnerabilities are discovered in the top 30% of vulnerable files.

This phenomenon raises a question whether it is because the top 30% of files account for the larger portion of code. Therefore, to measure the code complexity, we employ four metrics widely used in fault prediction [16, 50, 61]: non-blank lines of code (LoC), cyclomatic, maximum level of indentation of blocks within a region (MaxIndent) and the number of magic numbers. We extract the files



(a) Files

(b) Functions

Figure 4: Alberg diagrams showing the percentage of vulnerable files (functions) over the percentage of complexity for the projects. The x axis presents a descendingly sorted list of vulnerable files as per the number of vulnerabilities contained. The y axis is the cumulative percentage of code complexities based on four metrics.

consisting of vulnerabilities, and compute the four metrics. If one file has more than one vulnerability, we average the metric values for this file in terms of the number of vulnerabilities. From Figure 4a and Figure 3a, the vulnerability number and code complexity have a linear relation where the vulnerability number is monotonically increasing along with the increase of code complexity. Based on the above investigation, we draw the following finding.

Finding 1. *A small number (30%) of vulnerable files account for most (66%) of vulnerabilities. However, these files only contribute 40% of code size and complexities. Therefore, a higher complexity of code cannot necessarily induce more vulnerabilities, and complexity metrics in defect prediction cease to being effective for vulnerabilities.*

Same with the analysis of vulnerable files, we observe that these vulnerabilities do not obviously conform to the same phenomenon with regard to vulnerable functions, that is, top 30% vulnerable functions only contribute 50% vulnerabilities. Figure 3b shows the vulnerability number and function number exhibit a linear relation. In addition, Figure 4a depicts that the complexity for each function is relatively even with others. As a result, we conclude the spatial distribution with regard to vulnerable functions as follows.

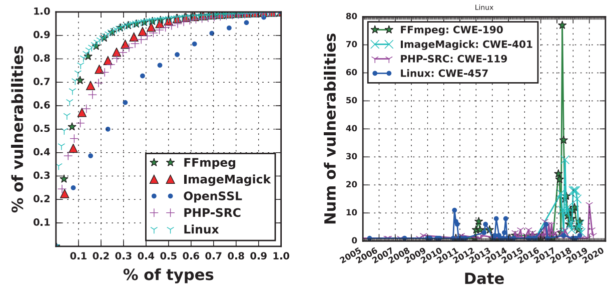
Finding 2. *With the consideration of vulnerable functions, top 30% vulnerable functions account for 50% vulnerabilities, not as obviously as vulnerable files do.*

3.2 Type Distribution

The type of vulnerabilities is labelled by NVD, following the CWE [35] classification system. As Section 2, we have scraped all meta information of vulnerabilities including type. However, by manually reviewing this information, we find that:

Finding 3. *There are 9.6% erroneous, 10.9% imprecise and 7.2% missing problems in type labelling of our investigated vulnerabilities, which have been reported to NVD.*

The three problems are detailed as follows.



(a) Alberg diagram showing the cumulative number of vulnerabilities over that of types.

(b) The number of vulnerabilities of the top 3 types changes over time.

Figure 5: Type distribution diagram

(1) **Erroneous:** Some CVE entires are assigned with wrong CWEs. For example, by examining the description and code of CVE-2009-2767 [36], it is sort of CWE-476 (Null Pointer Dereference). However, it is wrongly classified into CWE-119.
(2) **Imprecise:** There exist some CVE entries that are assigned with imprecise CWEs, *i.e.*, CWEs cannot describe their detailed classification. For example, CVE-2010-4250 [37] is bundled with CWE-399 (Resource Management Errors). However, with our analysis, it can be further classified into a child category CWE-401 (Memory Leak), which is more precise.
(3) **Missing:** Some CVE entries do not connect to any CWEs, such as CVE-2015-3636 [38].

We overcome the above problems by relying on a rigorous review and remediation by our professionals. For each project, we sort the types as per the number of contained vulnerabilities in descending order and calculate the cumulative percentage of vulnerabilities along with type. Figure 5a shows that 20% of types contain 70% of vulnerabilities. Besides, we list the top 3 types of each project as shown in Table 2. From this table, the 5 projects vary from the most 3 types. For FFmpeg, integer overflow (CWE-190) and numeric errors (CWE-189) are widely present. That is because FFmpeg is a multimedia encoding/decoding library, which originally involves a large number of arithmetic operations. For Linux, CWE-264 is among the top 3 types. It implies that privileges, permissions and access controls are the most important functionalities offered by an operating system kernel. Massive amounts of code are implemented to protect the system's resources and, however, induce a considerable number of vulnerabilities.

Finding 4. *For one C/C++ investigated project, 70% of vulnerabilities fall into 20% of types; the top 3 types account for nearly 50% of vulnerabilities averagely. Moreover, the most likely vulnerability types existing in one project largely depend on the project's core functionalities and programming languages.*

3.2.1 *Changes of Top Types over time.* How do the types with the most vulnerabilities change over time? We evaluate this temporal effect by counting the detected vulnerabilities of the top 3 types over years and present the change curve in Figure 5b. From the figure, we conclude that:

Table 2: The most 3 vulnerability types in each project.

Project	1st Type	2nd Type	3rd Type
FFmpeg	CWE-190	CWE-119	CWE-189
ImageMagick	CWE-401	CWE-119	CWE-457
OpenSSL	CWE-310	CWE-119	CWE-399
PHP-SRC	CWE-119	CWE-416	CWE-20
Linux	CWE-457	CWE-264	CWE-119

Finding 5. *The top types in a project are not dramatically affected by time, and will remain the majority in future. It implies that security analysts can spend prioritized efforts auditing vulnerabilities of the popular types based on the historical data.*

3.3 Developer Distribution

Vulnerability developer: *In this paper, we regard non-blank and non-comment lines deleted in a vulnerability-related commit as the vulnerable code, and for the commits that only insert lines of code, corresponding code will be marked manually. Developers creating such code are thereby responsible for the introduction of vulnerabilities.*

In this section, we make use of command "`git blame`" to identify the developer(s) for each vulnerability. For each developer, we count the number of vulnerable lines introduced or contributed by him/her. We sort the developers in a project as per the number of their introduced vulnerable lines in descending order.

From Figure 6, we can see all vulnerabilities of a project are introduced by less than 10% of its developers. It means that most of the developers seldom introduce any vulnerabilities. The code proportions contributed by vulnerability developers vary from 60% to nearly 100%. In other words, for a certain project, there exist such developers that contribute little proportion of code lines but introduce more vulnerabilities.

Finding 6. *Almost all of the vulnerabilities are introduced by less than 10% of developers for a project. Such 10% developers contribute from 60% to nearly 100% code, which varies cross projects depending on the number of their developers.*

3.3.1 Developers' Type Preference. Are developers prone to introducing vulnerabilities of the same type? If so, security analysts can try to discover new vulnerabilities of the same type in other projects to which the developer contributes. In the meantime, we can identify the inexperienced developers of secure programming. To this end, we compute the proportion of vulnerabilities with the top 1 type and the top 30% of types to all vulnerabilities introduced by a developer, shown in Figure 7a and 7b. For FFmpeg, 10% of developers introduce only one type of vulnerabilities, and the introduced vulnerabilities merely take up a small portion. There are 21 developers who introduce at least 10 vulnerabilities in project FFmpeg. Further, we find that they are of various types, and over 60% of them belong to the top 3 types. We conclude:

Finding 7. *Although we cannot speculate the type preferences of the developers who introduce < 10 vulnerabilities, it does exist in the remaining more productive developers. Therefore, this offers a guidance for prioritized code review in terms of the types of vulnerabilities likely to be introduced by a certain developer.*

4 RQ2: VULNERABILITY DEPENDENCY

Security researchers often find 0-day vulnerabilities in close proximity to 1-day vulnerabilities. For example, Jia *et al.* [28] found two new Microsoft Word vulnerabilities when generating the PoC of CVE-2014-1761. The illustrative example in Figure 1 also shows that this phenomenon is not just by accident. To explain why vulnerabilities often appear in near distance, we measure their distances and build the semantic relations between vulnerabilities, .

4.1 Vulnerability Distance

We have constructed vulnerability call graphs for each project and group vulnerabilities together in Section 2. Given one vulnerability call graph $\mathcal{G}\ (F, V, \delta)$ (as Definition 1) and two vulnerable functions v_1 and v_2, where $v_1, v_2 \in V$, the distance between v_1 and v_2 is computed by:

$$d(v_1, v_2) = \arg\min_n |\{f_1, ..., f_n\}| - 1$$

where $f_i \in F$, $f_1 = v_1$, $f_n = v_2$, and $(f_i, f_{i+1}) \in \delta$ or $(f_{i+1}, f_i) \in \delta$. The minimal value for n is 2, which means the two vulnerable functions have a direct call relationship in the call graph. If the distance is 2, there must be an intermediate function connecting the two vulnerable ones. We compute the distance by: first converting the directed vulnerability call graph into an undirected graph by eliminating the arrows of calls; then computing the shortest path for these two vulnerable functions in the graph.

In light of the distance between vulnerabilities, we can slice one call graph into small clusters. In particular, we define:

Definition 2. *A **k-cluster** C is a slice from the entire call graph G, while for all vulnerable functions* $v1, v2 \in C$, $d(v_1, v_2) <= k$, *and for every vulnerable function* $v' \in G/C$, $d(v_1, v') > k$.

We slice the VCG of a project into a number of 1-clusters and 2-clusters, respectively. Then we count the number of vulnerabilities in these clusters to check the percentage of vulnerabilities they contribute. We plot Figure 8a and 8b to demonstrate the contribution rates of 1-cluster and 2-cluster, respectively. In Figure 8a, only FFmpeg and ImageMagick have the phenomenon that 30% largest 1-clusters account for more than 60% vulnerabilities.

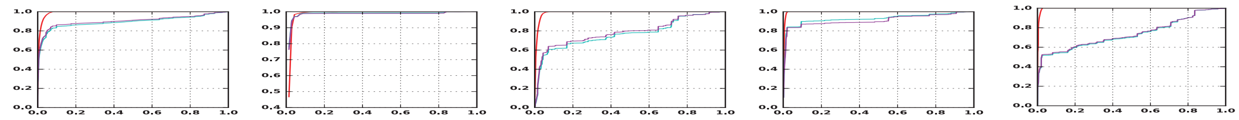
But for the other 3 projects, such clustering phenomenon is not obvious. Moreover, we can see that about 20% of all functions can be sliced into *1-clusters* for a project. As for 2-cluster, the largest 2-cluster in the projects consists of 80% - 100% of the vulnerabilities. Based on the analysis, we conclude:

Finding 8. *The locations where vulnerabilities emerge are very concentrated. For each investigated project, more than 80% of vulnerable functions assemble in one single 2-cluster.*

4.2 Patterns of Dependency

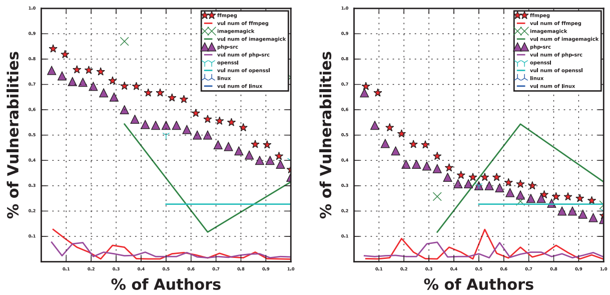
In the previous section, we obtain a primary 2-cluster for each project. The primary 2-cluster contains 80% - 99% of vulnerable functions, however, it accounts for about 50% of all functions of a project. In order to further study the relations, we formulate four patterns of dependency relations:

- **Common Parent:** two vulnerable functions have a common predecessor node (function) in the call graph. The predecessor function can be either vulnerable or non-vulnerable.



(a) FFmpeg. (b) ImageMagick. (c) PHP-SRC. (d) OpenSSL. (e) Linux.

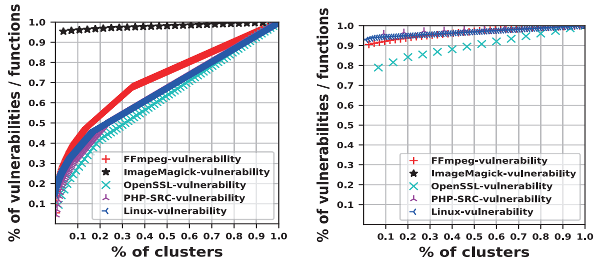
Figure 6: Alberg diagrams showing the percentage of LOCs over the percentage of all developers for the projects. Red line represents vulnerable lines introduced by developers.



(a) Num. of vulnerabilities with 30% types versus developers for each project.

(b) Num. of vulnerabilities with the most type versus developers for each project.

Figure 7: Developer distribution diagram



(a) k=1 (b) k=2

Figure 8: Alberg diagrams showing the percentage of vulnerabilities over the percentage of k-clusters for each project

- **Common Child:** two vulnerable functions have a common successor node (function) in the call graph. Here we consider the pattern only when the successor is also vulnerable.
- **One Jump Call:** one vulnerable function calls an intermediate non-vulnerable function which has a direct call to the other vulnerable function.
- **Direct Call:** one vulnerable function has a direct call to the other.

One vulnerability-related commit may alter multiple functions so that these functions are all tainted with the vulnerability.

Finding 9. *More than 60% vulnerable functions of a project have at least one basic call-relation to another vulnerable function with different vulnerabilities.*

As shown in Figure 10, for each project, we count the number of vulnerable function pairs according to each basic dependency relation. Further, we calculate the intersection of four relation sets in which an element is a vulnerable function pair. For example, in FFmpeg, there are 233 vulnerable function pairs that have *common parent* relation; among these, three pairs also have *common child* relation, another three pairs have *direct call* relation, two pairs have *one jump* relation, one pair has *direct call* and *one jump* relation at the same time. Besides, we count the total number of vulnerable functions in the union of four dependency relation sets. For example, 72% vulnerable functions have at least one basic call relation with another vulnerable function and they have different vulnerabilities.

In all, for more than 60% vulnerable functions, there exists another vulnerable function in the call graph between which the distance is no more than 2. It inspires that one is likely to discover new vulnerabilities in a function that has the above types of dependency relations with the already-known vulnerable functions.

4.3 Semantics of Dependency

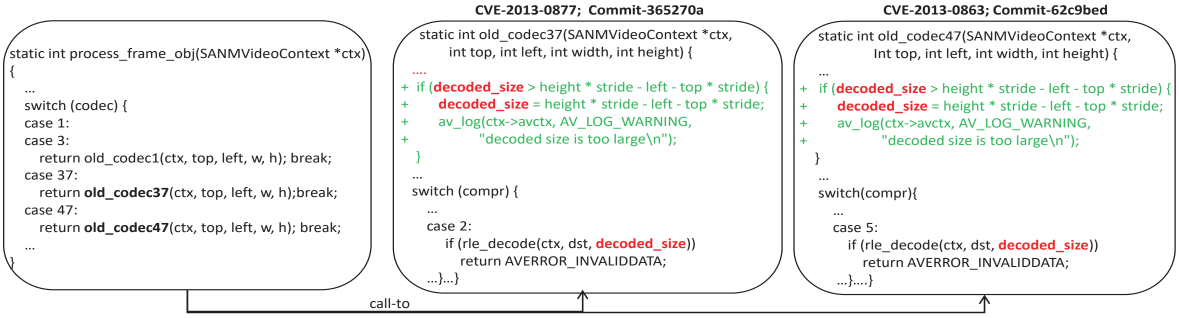
In this section, we intend to identify what code structure and relations exist between vulnerable code blocks.

4.3.1 Manual Analysis. Given a vulnerable function pair (v_1, $v2$) complying with one of the four aforementioned dependency patterns, three experts spent six person months to conduct a rigorous and manual code review procedure, in order to determine the correlations between vulnerable code in v_1 and that in v_2.

For example, in Figure 9, CVE-2013-0863 resides in the function `old_codec47` and CVE-2013-0877 occurs in `old_codec37`. Both vulnerable functions are called by a *common parent*. By reviewing the code at call sites, we determine that there are four types of relations between them: (1) these vulnerable functions are in two parallel switch-case statements to decode different video types; (2) both vulnerabilities are caused by the same missing check on the third parameter of function `rle_decode`, *i.e.*, they have similar operations; (3) both vulnerability are related to the same key variable *decode_size*, which holds some values read from input; and (4) they both deal with multimedia files of *SANM* format.

4.3.2 Semantic Relations. Finally, we identify 10 types of semantic relations between vulnerable code as follows.

- **R1:** two vulnerable functions are parallel from the view of control flow. In particular, two functions may be called from different `cases` inside a `switch` statement, such as CVE-2013-0845 (FFmpeg: 0ceca26) and CVE-2012-2775 (ffmpeg: 9d3032b).
- **R2:** vulnerable code in two functions executes similar or same operations. e.g., CVE-2013-0863 and CVE-2013-0877.
- **R3:** vulnerability-related code in two functions has similar or same code structure in syntax, such as OSS-Fuzz issue NO.1903 [23] (FFmpeg: 58f8cd4) and NO.1506 [22] (FFmpeg: 5ac17f1).



```
static int process_frame_obj(SANMVideoContext *ctx)
{
   ...
   switch (codec) {
   case 1:
   case 3:
      return old_codec1(ctx, top, left, w, h); break;
   case 37:
      return old_codec37(ctx, top, left, w, h);break;
   case 47:
      return old_codec47(ctx, top, left, w, h); break;
   ...
}
```

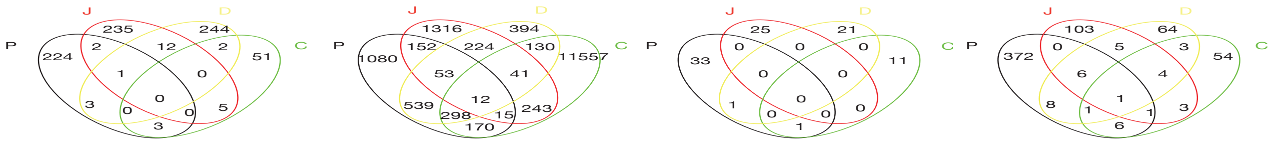
CVE-2013-0877; Commit-365270a

```
static int old_codec37(SANMVideoContext *ctx,
      int top, int left, int width, int height) {
 ....
+  if (decoded_size > height * stride - left - top * stride) {
+      decoded_size = height * stride - left - top * stride;
+      av_log(ctx->avctx, AV_LOG_WARNING,
+             "decoded size is too large\n");
   }
 ...
 switch (compr) {
   ...
   case 2:
      if (rle_decode(ctx, dst, decoded_size))
         return AVERROR_INVALIDDATA;
   ...}...}
```

CVE-2013-0863; Commit-62c9bed

```
static int old_codec47(SANMVideoContext *ctx,
           Int top, int left, int width, int height) {
  ...
+ if (decoded_size > height * stride - left - top * stride) {
+     decoded_size = height * stride - left - top * stride;
+     av_log(ctx->avctx, AV_LOG_WARNING,
+            "decoded size is too large\n");
  }
  ...
  switch(compr){
    ...
    case 5:
       if (rle_decode(ctx, dst, decoded_size))
          return AVERROR_INVALIDDATA;
    ...}....}
```

Figure 9: Example: Cross-function Semantic Relation Manual Analysis.



(a) FFmpeg

(b) ImageMagick

(c) OpenSSL

(d) PHP-SRC

Figure 10: Venn diagrams showing the statistics of dependency patterns. An element is a pair of vulnerable functions that follows some Call-Relation Unit.

- **R4:** the variables involved in vulnerabilities have data flow relation in between. e.g., OSS-Fuzz issue NO.7420 (FFmpeg: dce80a4) and NO.4415 (FFmpeg: 1d0817d).
- **R5:** both the functions involve the same variable(s). e.g., CVE-2015-6818 and CVE-2017-7863.
- **R6:** the variables involved by vulnerabilities have control flow relation in between. e.g., OSS-Fuzz issue 2873 (ffmpeg: 6f03ffb) and 3444 (ffmpeg: dcf9bae).
- **R7:** vulnerability-related code in two functions are under the same macro condition. For example, OSS-Fuzz issue 1471 (ffmpeg: 3a0ff78) and 1878 (ffmpeg: 6b9cb5d).
- **R8:** two functions are used to deal with the same data formats. e.g., CVE-2013-0863 vs. CVE-2013-0877.
- **R9:** two functions are patched in one commit. e.g., OSS-Fuzz issue 5894 (FFmpeg: 647fa49).
- **R10:** two functions are called in the same case of a switch. e.g., OSS-Fuzz issue 3444 (3a0ff78) and 2581 (2886142).

Figure 11 shows an illustrative example about these semantic relations in a 2-cluster from project FFmpeg. Function aac_decode_frame_int has **R1** relation with function decode_ics. The code shows the vulnerability of aac_decode_frame_int occurs within a case block of a switch statement, while function decode_ics is called from another case block. So the two vulnerabilities are parallel in the view of control flow. Both the vulnerabilities "OSS-Fuzz issue 1471" in subband_scale and "OSS-Fuzz issue 1878" in noise_scale are caused by an overlarge left-shift offset, inducing integer overflow. So these two functions have **R2** relation. At the same time, related code blocks of these two vulnerabilities are similar in syntax, indicating **R3** relation. Besides, subband_scale and noise_scale are both called in function decode_spectrum_and_dequant, and the vulnerability-related variable in subband_scale has a data flow to the vulnerability-related variable in noise_scale, which denotes **R4** relation. There is no data flow between function apply_dependent_coupling_fixed and apply_independent_coupling_fixed. But two vulnerabilities "OSS-Fuzz issue 1851" and "issue 1762" in these functions manipulate the same variable of the parameter object, so they have **R5** relation.

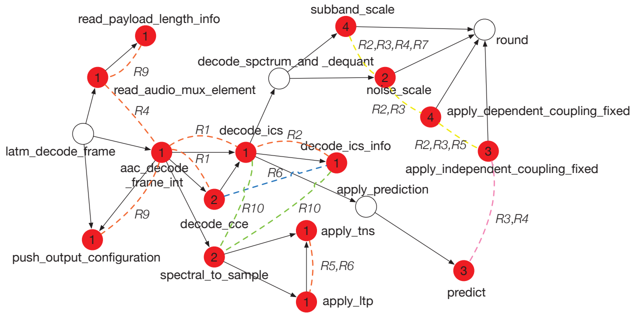


Figure 11: A simplified 2-cluster example from project FFmpeg with multiple semantic relations between functions. A red solid circle represents a vulnerable function with a number inside counting the contained vulnerabilities; A hollow circle represents a non-vulnerable function; A black arrow represents the call relation between functions; A dotted line with a label represents semantic relations between them.

Because the analysis is time-consuming and labor-intensive, we just took FFmpeg as an example and sampled 324 vulnerable function pairs with these semantic relations, According to the results summarized in Table 3, most (70.5%) of vulnerable function pairs

Table 3: Inter-function semantic relations underlying the four patterns from project FFmpeg. Notedly, a pair of vulnerable functions may possess multiple relations.

Relation	# Common Parent	# Common Child	# Direct Call	# One Jump	# Total
R1	64	0	4	5	73
R2	86	10	13	4	113
R3	33	1	7	1	42
R4	42	5	14	11	72
R5	48	4	9	5	66
R6	25	10	21	21	77
R7	6	0	1	1	8
R8	170	19	36	16	241
R9	6	2	7	0	15
# Total of Pairs	241	19	40	24	

are due to processing the same file format (*i.e.*, **R8**). For **Common-Parent** relation, 26.6% of function pairs have Parallel Control Flow relation, while pairs with the other three relations rarely have **R2** relation. For vulnerabilities from **Common-Parent** and **Common-Child** pairs, **R2** is the most common. While for vulnerabilities from **Direct-Call** and **One-Jump** pairs, **R6** is the most common.

5 RQ3: VULNERABILITY RECURRENCE

It is observed that some functions are constantly found with new vulnerabilities and hence have undergone several security fixings. We call them high-frequency vulnerable functions and in this section, we compute their statistics, measure the duration between two disclosed vulnerabilities, and demystify the primary reasons of this recurrence phenomenon.

5.1 High-frequency Vulnerable Functions

For all target projects, on average 25.5% of functions have been discovered with at least 2 vulnerabilities.

Statistics. For each project, we compute the proportion of such functions in the set of vulnerable functions, shown as Column 2 in Table 4. The proportions of high-frequency vulnerable functions vary from 12.7% (OpenSSL) to 38.3% (ImageMagick). In addition, the average, median, maximal and minimal numbers of vulnerable functions are also presented from Column 3 to 6. More specifically, function `mpeg4_decode_sprite_trajectory` in FFmpeg has been found with 13 vulnerabilities. The function `SPL_METHOD(Array,unserialize)` has been found with 8 vulnerabilities.

Duration. We investigate the time window during which two successive vulnerabilities are discovered in the same function, and present the result in Table 5. For OpenSSL, the average time interval between the disclosure dates of two successive vulnerabilities in the same function is 1024.16 days, and the median value is 671.57 days. However, for ImageMagick, the average time interval is 65.62 days and the median is only 9.54 days. On average, the recurrence phenomenon happens in a function with a 514.95-day interval. It is worth mentioning that the time interval can be 0. This is because multiple vulnerabilities are sometimes patched in the same commit, *e.g.*, CVE-2013-7267 and CVE-2013-7266.

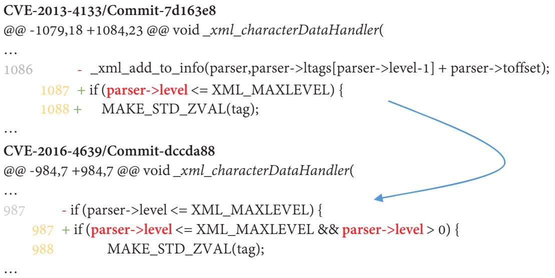
Finding 10. *The recurrence phenomenon occurs in a considerable number (25.5%) of functions with 3 times on average. However, the interval between two security fixings can be up to 514.95 days.*

Table 4: Statistics of high-frequency vulnerable functions.

Project	% of high-frequency functions	Avg.	Median	Max.	Min.
FFmpeg	27.1	2.59	2	13	2
ImageMagick	38.3	4.85	3	36	2
OpenSSL	12.7	2.59	2	6	2
PHP-SRC	17.2	2.62	2	8	2
Linux	12.8	2.37	2	8	2
Average	**25.5**	**3.00**	**2.2**	**14.2**	**2**

Table 5: Statistics of time interval when two successive vulnerabilities are discovered in a same function.

Project	Avg. (day)	Median (day)	Max. (day)	Min. (day)
FFmpeg	191.52	29.28	2245.15	0.0
ImageMagick	65.62	9.54	1123.38	0.0
OpenSSL	1024.16	671.57	4612.09	0.0
PHP-SRC	496.87	189.62	4005.45	0.0
Linux	796.57	457.29	4661.33	0.0
Average	**514.95**	**271.46**	**3329.48**	**0.0**



```
CVE-2013-4133/Commit-7d163e8
@@ -1079,18 +1084,23 @@ void _xml_characterDataHandler(
...
1086        - _xml_add_to_info(parser,parser->ltags[parser->level-1] + parser->toffset);
     1087 + if (parser->level <= XML_MAXLEVEL) {
     1088 +     MAKE_STD_ZVAL(tag);
...
CVE-2016-4639/Commit-dccda88
@@ -984,7 +984,7 @@ void _xml_characterDataHandler(
...
987       - if (parser->level <= XML_MAXLEVEL) {
      987 + if (parser->level <= XML_MAXLEVEL && parser->level > 0) {
      988           MAKE_STD_ZVAL(tag);
...
```

Figure 12: Vulnerability recurrence example in PHP-SRC

5.2 Primary Reasons

What's the reason behind the recurrence phenomenon and what are the semantic relations between vulnerabilities within the same function? To answer these questions, we manually analyze 1,095 vulnerabilities occurred in 348 high-frequency vulnerable functions from project FFmpeg, OpenSSL and PHP-SRC and ImageMagick.

5.2.1 *Manual Analysis.*

Three analysts were designated to identify the causes of the recurrent phenomenon and the intra-function relations between vulnerabilities. Taking Figure 12 as an example, two vulnerabilities were found in one function `_xml_characterDataHandler`. The first vulnerability (CVE-2013-4113), which causes heap corruption, was patched with a sanity check in Line 1087. However, this sanity check has been proved incomplete, since a heap buffer over-read could happen if the integer variable `parser->level` is zero or less. By using a customized code change tracker based on Git (Section 2.4.2), we find that another commit dccda88 fixed this problematic patch. After that, one analyst determines the cause of the latter vulnerability as incomplete fix.

5.2.2 *Semantic Relations.*

We spend 3 person months identifying the following semantic relations between vulnerabilities within the same function:

- **R1: Incomplete fix or fix introduction.** Another vulnerability occurs because the fix to the former vulnerability is incomplete or it introduces new risks. e.g., CVE-2016-6307 (openssl: acacbfa) vs. CVE-2016-6309 (openssl: 4b390b6).

- **R2: Same or Similar Vulnerability Pattern**. Two vulnerabilities have similar or same vulnerability pattern. e.g., both CVE-2015-6835 (php-src: df4bf28) and CVE-2016-6290 (php-src: 8763c60) are induced because of improper exception handler.
- **R3: Same or Similar Code Structure**. The code blocks where vulnerabilities exist are same or similar in syntax. e.g., CVE-2014-3515 (php-src: 88223c5) vs. CVE-2016-7417 (php-src: ecb7f58).
- **R4: Involved Same Variable.** Two vulnerabilities are directly or indirectly related with the same variable. For example, CVE-2019-9022 (php-src: 8d3dfab) vs. CVE-2014-4049 (php-src: b34d784).
- **R5: Parallel Control Flow.** Related code is located in parallel branches, such as different switch-cases. e.g., CVE-2012-0852 (ffmpeg: 6087080) vs. CVE-2013-0844 (ffmpeg: f18c873)

Table 6 lists the number of detected relations existing between vulnerabilities in the four projects. Based on the result, we find:

Finding 11. *Incomplete fix and fix introduction are the major reason why vulnerabilities repeatedly occur in the same function. It implies that some developers likely fail to fix all existing vulnerabilities or make new ones stemming from the patching code.*

This problem happens many times as observed in our dataset, for instance CVE-2016-6307 (OpenSSL: acacbfa) vs. CVE-2016-6309 (OpenSSL: 4b390b6) and CVE-2013-4113 (PHP-SRC: 7d163e8) vs. CVE-2016-4539 (PHP-SRC: dccda88). It raises an alarm for the developers that they need to pay more attention to the security of patches, avoiding introducing additional vulnerabilities. In the meantime, this finding can motivate researchers to analyze and evaluate the security of patches, just as UC-Klee [43].

We also observe that **R2** occurs with the second most frequency. This can explain why works like SPAIN [59] can be effective in detecting similar vulnerabilities to some extent. Different from SPAIN, which focuses on discovering inter-function similar vulnerabilities, we observe that there exist many similar vulnerabilities within one function. This conclusion has at least two benefits: on one hand, it stresses the importance of research on similar or cloned vulnerabilities [31]. On the other hand, it can be used by security researchers to quickly detect other vulnerabilities in a cost-effective way.

Moreover, we observe that **R3** occurs in all of the projects. If two blocks of code are of a high similarity, they can be easily matched by code clone techniques intuitively. However, according to our investigation, developers prefer not to perform such a code search and merely fix the reported vulnerabilities. More specifically, when a developer tries to fix a reported vulnerability, she/he only locates the root causes perhaps with PoCs and then makes all the necessary changes to code to eliminate the reported vulnerabilities. But the developer fails to audit other similar or cloned code that may contain similar vulnerabilities.

It sheds light on the necessity of a robust vulnerability response process, from where software engineering practitioners can research to improve the process, figure out more vulnerability detection solutions and empirically study the features of vulnerabilities.

Finding 12. *There exists a "Report One and Patch One" (ROPO) problem commonly in vulnerability fixing. It reveals one weakness of the vulnerability fixing process, and inspires many future research directions such as the enhancement of patching process and efficient detection of similar or cloned vulnerabilities.*

Table 6: Intra-function semantic relations of vulnerabilities within one function

Relation	# FFmpeg	# ImageMagick	# OpenSSL	# PHP-SRC
R1	83	53	6	40
R2	66	32	5	16
R3	9	6	3	10
R4	39	8	0	10
R5	4	9	0	7

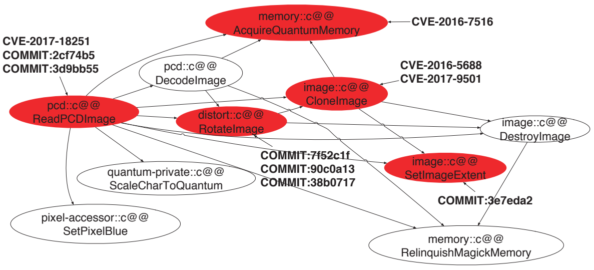


Figure 13: Illustrative graph for new vulnerability detection

6 APPLICATION OF FINDINGS

Our findings have a great potential for hunting new vulnerabilities. To evaluate their efficacy, we implement two proof-of-concepts that integrate the above findings.

Guided Code Review. We take ImageMagick as an example. According to Finding 5, the type CWE-401 (Memory Leak) consists of the largest number of vulnerabilities, and it sustainedly contributes the majority of vulnerabilities. Hence we prioritize our efforts to explore new Memory Leak vulnerabilities. Further, according to our observation, most of Memory Leak vulnerabilities in ImageMagick are caused by missing the memory freeing operation in exception handlers. We implement a prototype to search similar patterns. Finding 8 implies that most of vulnerable functions are adjacent with each other in the vulnerability call graph. So we prioritize our review targets to such functions that have dependency relations (*e.g.*, Common Parent) with known vulnerable functions. Figure 13 illustrates just the case. Function `DecodeImage` has **Common Parent**, **Common Child** and **One Jump** relation with other vulnerable functions. Using these findings, we successfully found a new Memory Leak vulnerability (CVE-2019-7175 [1]) in `DecodeImage`. Moreover, we found another 4 new vulnerabilities (*CVE-2019-7395, CVE-2019-7396, CVE-2019-7397,CVE-2019-7398*) in a similar way.

Directed Fuzzing. We observed that the *k-cluster* phenomenon is very common in vulnerability distribution as stated in Finding 8. We enhance the AFL by directing the seed selection and prioritizing mutation strategy. Testing on FFmpeg, we first get the largest *2-cluster* and then use this information to select better seeds. In addition, we modify the priority strategy and assign the functions in the 2-cluster the with higher priorities. Finally, we discovered five new vulnerabilities (*Commit-3713833, CVE-2017-11399, CVE-2017-11719, CVE-2017-14767, CVE-2017-16840*) of project FFmpeg.

In all, we have undertaken responsible disclosure, reported 10 vulnerabilities to the related vendors and finally got 9 new CVE numbers from these target projects. These two proof of concepts above show that our findings can facilitate the vulnerability detection to a large extent.

7 RELATED WORK

Vulnerability Analysis. Shahzad *et al.* [48] characterized vulnerabilities in a wide range of aspects including the phases when the vulnerabilities were introduced, how they evolved over the years, and functionalities of vulnerabilities. Ruohomen conducted a release-based time series analysis for Python vulnerabilities [44], and found the appearance probabilities of vulnerabilities in different versions obeyed the Markov property. Kim and Lee [30] analyzed the cloned vulnerabilities and summarized the corresponding root causes as well as their life cycle features. By examining the response from 21 software vendors to 241 vulnerabilities, Arora *et al.* [11] studied the influential factors to the timeliness of security patches and unveiled the relationship between competition and software quality. Li and Paxson investigated over 4,000 security patches for 3,000+ vulnerabilities and unveiled the patching characteristics with regard to the development life cycle [32]. *Apart from vulnerability distribution, our study focuses on unveiling the semantic relations between vulnerabilities. The analysis results can explain why vulnerabilities occur and further assist in vulnerability detection.* Gkortzis et al. [21] framed a vulnerability dataset from open-source systems, and proposed several metrics to characterize these vulnerabilities. *Besides vulnerability code and associated properties, our dataset is also populated with many mature yet handy features including vulnerability introducer and call relations.*

Vulnerability Prediction. Shin and Williams [49] unveiled that the fault prediction model and the vulnerability prediction model (VPM) provided similar prediction performance [51]. Zimmermann *et al.* [63] found that software measures could only achieve a high precision rate but low recall rate in vulnerability prediction. Doyle and Walden [13] observed a trend of decreasing vulnerability density over time. Morrison *et al.* [39] concluded that VPMs must be refined to achieve actionable performance. Scandariato *et al.* [46] applied text mining source code to predict vulnerable software components. Perl *et al.* built a mapping from the CVEs and vulnerability-contributing commits and developed an SVM-based classifier to identify potential vulnerabilities in a large code base [42]. *Our study concludes many significant findings and insights that can aid both static- and dynamic- analysis based vulnerability prediction (cf. Section 6).*

8 DISCUSSION

Threats to validity. *Data quality of vulnerabilities.* Vulnerability entries in NVD/CVE repos contain many human-crafted pieces of information such as vulnerability type, description, and references to vulnerable code. Therefore, errors and inaccuracies inevitably flow in and degrade the quality of data. Our manual auditing cannot ensure a complete solution for solving this; We determine the position of vulnerabilities by checking what code is deleted in a commit or influenced by the added code. However, it is non-trivial to identify the exact scope of vulnerable code, and any inaccurate labeling can have a negative impact on the analysis results. Additionally, we did not consider the changes to a pure data structure or variable, which take less than 1% in our dataset. Instead, we concentrate on the changes that alter actional code.

Generalization limits. Although our analysis is built on a large number (3,938) of vulnerabilities from five popular projects with varying functionalities, we cannot make our findings as a general claim. On one hand, our target projects are limited to C/C++ projects. The findings may not apply for projects written in other languages. For example, buffer overflow (CWE-119) is never a security issue in Java projects. On the other hand, we only analyzed 5 projects due to the overwhelming manual work. It could be insufficient to make a general claim from only 5 projects' observations. In all, while there is an indication that these findings may persist across C/C++ projects, further studies on larger project populations written in various languages are required to draw a general claim.

Future research direction. Our study as well as our findings can inspire several future research directions. For example, the semantic relations between vulnerabilities can facilitate the understanding of their root causes; the dependency patterns which reveal the call relation and distance can power static analysis techniques of more efficiency; the ROPO problem in Finding 12 motivates researchers to re-inspect the patching process, keep more attention on the security of patches, and work out a better solution.

9 CONCLUSION

We conduct a large-scale empirical study of security vulnerabilities, evaluating 3,938 vulnerabilities from 5 popular projects. First the vulnerabilities are attributed with descriptive information (*e.g.*, type, involved developer), and blamed code. Then we leverage a lightweight static analysis to build the calling connections between vulnerabilities and extracted code semantics to represent them. We have extensively characterized vulnerabilities from three aspects: distribution, dependency and recurrence, and distilled 12 findings that are beneficial for the future research. Guided by our findings, we developed proof-of-concepts and successfully discovered 10 zero-day vulnerabilities.

ACKNOWLEDGMENTS

We would thank Defang Bo, Lulu Tian, Zhaoming Jin and Zheming Li for their help in data processing and paper writing. We would also thank the anonymous reviewers for their valuable comments. The work is supported by the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology, as well as National Natural Science Foundation of China under Grant No.: 61772308, 61972224, U1736209, 61802394, U1836209, 61902395; National Key Research and Development Program of China under Grant No.2016QY071405; Strategic Priority Research Program of CAS under Grant No.: XDC02040100, XDC02030200, XDC02020200; BNRist Network and Software Security Research Program under Grant No.BNR2019TD01004 and No.BNR2019RC01009.

REFERENCES

[1] [n.d.]. *CVE-2019-7175.* Retrieved February 4, 2020 from https://github.com/ImageMagick/ImageMagick/issues/1450

[2] [n.d.]. *FFmpeg.* Retrieved August 15, 2019 from https://ffmpeg.org/

[3] [n.d.]. *ImageMagick.* Retrieved August 15, 2019 from https://imagemagick.org/

[4] [n.d.]. *ImageTragick.* Retrieved August 15, 2019 from https://imagetragick.com/

[5] [n.d.]. *Linux-kernel.* Retrieved August 15, 2019 from https://www.kernel.org/

[6] [n.d.]. *OpenSSL.* Retrieved August 15, 2019 from https://www.openssl.org/

[7] [n.d.]. *PHP-SRC.* Retrieved August 15, 2019 from https://www.php.net

[8] 2010. *american fuzzy lop.* Retrieved March 09, 2019 from http://lcamtuf.coredump.cx/afl/

[9] Amritanshu Agrawal and Tim Menzies. 2018. Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In *Proceedings of the 40th International Conference on Software engineering.* ACM, 1050–1061.

[10] Carina Andersson and Per Runeson. 2007. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering* 33, 5 (2007), 273–286.

[11] Ashish Arora, Chris Forman, Anand Nandkumar, and Rahul Telang. 2010. Competition and patching of security vulnerabilities: An empirical analysis. *Information Economics and Policy* 22, 2 (2010), 164–177. https://doi.org/10.1016/j.infoecopol.2009.10.002

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2329–2344. https://doi.org/10.1145/3133956.3134020

[13] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture* 57, 3 (2011), 294–313.

[14] Giulio Concas, Michele Marchesi, Alessandro Murgia, Roberto Tonelli, and Ivana Turnu. 2011. On the distribution of bugs in the eclipse system. *IEEE Transactions on Software Engineering* 37, 6 (2011), 872–877.

[15] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368* (2017).

[16] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering (ICSE).* 60–71. https://dl.acm.org/citation.cfm?id=3339514

[17] Norman E. Fenton and Niclas Ohlsson. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software engineering* 26, 8 (2000), 797–814.

[18] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. 2006. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense.* ACM, 131–138.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP).* 679–696. https://doi.org/10.1109/SP.2018.00040

[20] GitHub. [n.d.]. *GitHub Developer API.* Retrieved August 15, 2019 from https://developer.github.com/v3/

[21] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018.* 18–21. https://doi.org/10.1145/3196398.3196454

[22] Google. [n.d.]. *OSS-Fuzz issue 1506.* Retrieved August 15, 2019 from https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1506

[23] Google. [n.d.]. *OSS-Fuzz issue 1903.* Retrieved August 15, 2019 from https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1903

[24] Tihana Galinac Grbac, Per Runeson, and Darko Huljenić. 2012. A second replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering* 39, 4 (2012), 462–476.

[25] Tihana Galinac Grbac, Per Runeson, and Darko Huljenić. 2016. A quantitative analysis of the unit verification perspective on fault distributions in complex software systems: an operational replication. *Software quality journal* 24, 4 (2016), 967–995.

[26] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 618–635.

[27] James W. Hunt and M. Douglas McIlroy. 1976. *An Algorithm for Differential File Comparison.* Technical Report.

[28] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. 2017. Towards efficient heap overflow discovery. In *26th {USENIX} Security Symposium ({USENIX} Security 17).* 989–1006.

[29] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2016. Vulnerability prediction models: A case study on the linux kernel. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 1–10.

[30] Seulbae Kim and Heejo Lee. 2018. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Computers & Security* 77 (2018), 720–736. https://doi.org/10.1016/j.cose.2018.02.007

[31] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* 595–614. https://doi.org/10.1109/SP.2017.62

[32] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2201–2215.

[33] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* 627–637. https://doi.org/10.1145/3106237.3106295

[34] MITRE. [n.d.]. *Common Vulnerabilities and Exposures.* Retrieved August 15, 2019 from https://cve.mitre.org

[35] MITRE. [n.d.]. *Common Weakness Enumetation.* Retrieved August 15, 2019 from https://cwe.mitre.org

[36] MITRE. [n.d.]. *CVE-2009-2767.* Retrieved August 15, 2019 from https://www.cvedetails.com/cve/CVE-2009-2767

[37] MITRE. [n.d.]. *CVE-2010-4250.* Retrieved August 15, 2019 from https://www.cvedetails.com/cve/CVE-2010-4250

[38] MITRE. [n.d.]. *CVE-2015-3636.* Retrieved August 15, 2019 from https://www.cvedetails.com/cve/CVE-2015-3636

[39] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. 2015. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security.* ACM, 4.

[40] U.S. National Institute of Standards and Technology. [n.d.]. *National Vulnerability Database (NVD).* Retrieved August 15, 2019 from https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=last3years

[41] Andy Ozment and Stuart E Schechter. 2006. Milk or wine: does software security improve with age?. In *USENIX Security Symposium.* 93–104.

[42] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 426–437. https://doi.org/10.1145/2810103.2813604

[43] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 49–64.

[44] Jukka Ruohonen. 2018. An Empirical Analysis of Vulnerabilities in Python Packages for Web Applications. In *9th International Workshop on Empirical Software Engineering in Practice IWESEP*. 25–30. https://doi.org/10.1109/IWESEP.2018.00013

[45] Robert Sanders. 1987. The Pareto principle: its use and abuse. *Journal of Services Marketing* 1, 2 (1987), 37–40.

[46] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[47] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 771–781.

[48] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 771–781. https://doi.org/10.1109/ICSE.2012.6227141

[49] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2010), 772–787.

[50] Yonghee Shin and Laurie Williams. 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 315–317. https://doi.org/10.1145/1414004.1414065

[51] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* 18, 1 (2013), 25–59.

[52] CK Shriram, K Muthukumaran, and NL Bhanu Murthy. 2018. Empirical Study on the Distribution of Bugs in Software Systems. *International Journal of Software Engineering and Knowledge Engineering* 28, 01 (2018), 97–122.

[53] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 321–332.

[54] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[55] Dimitri van Heesch. [n.d.]. *Doxygen.* Retrieved July 1, 2019 from http://www.doxygen.nl/

[56] Wikipedia. [n.d.]. *DirtyCow.* Retrieved August 15, 2019 from https://en.wikipedia.org/wiki/Dirty_COW

[57] Wikipedia. [n.d.]. *Heartbleed.* Retrieved August 15, 2019 from https://en.wikipedia.org/wiki/Heartbleed

[58] Wikipedia. [n.d.]. *SandWorm.* Retrieved August 15, 2019 from https://www.cvedetails.com/cve/CVE-2014-4114

[59] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 462–472.

[60] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. 590–604. https://doi.org/10.1109/SP.2014.44

[61] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 157–168. https://doi.org/10.1145/2950290.2950353

[62] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2011. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*. ACM, 93–102.

[63] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 421–428.