Vulnerability Analysis of Similar Code

Azin Piran
New York Institute of Technology
Vancouver, BC, Canada
apiran@nyit.edu

Che-Pin Chang
New York Institute of Technology
Vancouver, BC, Canada
cchang27@nyit.edu

Amin Milani Fard
New York Institute of Technology
Vancouver, BC, Canada
amilanif@nyit.edu

***Abstract*—Studying frequent code vulnerabilities in similar code, such as clones, near-duplicates, forked projects, or libraries, can help in the automated detection of security flaws during the software development process. In this work we conduct an empirical study on vulnerabilities in C/C++ code to characterize security flaws and find out if the same vulnerabilities exist in applications that share similar code or have the same business logic/domain. We analyze a code vulnerability dataset including 315 projects with 3284 security issues in 10,880 functions. Our results show that vulnerable functions in 35% of the most occurring CWEs (software weaknesses types) have similar code, and 23% of projects with the same domain/category have same the vulnerabilities. We observe that the most prevalent vulnerabilities in similar code are Use After Free, Improper Access Control, Cryptographic Issues, 7PK - Security Features, DoubleFree, Cross-site Scripting, and Divide By Zero. These vulnerabilities are, however, less frequent compared to other CWEs across all subjects. Our results suggest that automated vulnerability detection tools that work based on code similarity or abstract patterns can be tailored more towards certain CWEs.**

***Index Terms*—Code vulnerability, static analysis, CWE, CVE**

I. Introduction

Software security is pivotal in the development process of many real-world applications and hence notifying developers of possible code security issues can be very beneficial. For instance the GitHub Advisory Database [2] can provide details about the vulnerabilities to inform developers about security updates on package dependencies. Unfortunately, security is often seen as something that hinders software functionality development, and in most case it is not a primary skill or interest of software developers. Thus, software development teams should either have security analysts with solid coding knowledge or utilize code security analyzer tools. Existing vulnerability detectors often apply pattern matching based on rule-sets and have limited scope and coverage. There exist a few work on detecting vulnerabilities using code security characteristics datasets [5], [14], [17], [23]. Studies on similar code fragments in large applications including Mozilla [22] and Android applications [27] indicate the existence of vulnerability of some similar code fragments in different versions.

Code reuse for performing trivial tasks is widespread and a large portion of software today contains clones [18], [25] that can negatively influence the code maintainability and debugging [6], [21], or result in license violations [11]. Studying possible vulnerabilities in *"similar code"* or *"similar application logic"* can help developers write better code with fewer security flaws.

We study the existence of vulnerabilities in projects that share similar code, such as clones, near-duplicates, forked projects, and libraries, or has the same application logic/domain, which can be stated as follows: Let $P = \{p_1, ..., p_n\}$ be a set of projects, and $V = \{v_1, ...,v_m\}$ be a set of vulnerability classes. If p_i is similar to p_j, and p_i has a vulnerability in v_k, does p_j have a vulnerability in v_k? For instance, consider the *Stack-based Buffer Overflow* vulnerability. Cloning a code snippet that instantiates an array and accesses it in a loop, always presents this weakness unless the loop is bounded to the size of that array.

Our goals are (1) to identify the most common security flaws that can pertain to code reuse, and (2) to investigate whether existing vulnerabilities of an application can be used to locate the same vulnerabilities in applications that share similar code. We perform our study across a dataset of 315 open source C/C++ projects and conduct empirical analyses to address the following research questions:

- ***RQ1:*** *Are code with the same vulnerabilities similar?*
- ***RQ2:*** *Do the same vulnerabilities occur within projects of the same category/domain?*
- ***RQ3:*** *Which vulnerabilities are more prevalent in similar code?*

II. Study Design

A. Assumptions

Similarity. Code fragment redundancy and near-duplicates can be due to copy-and-paste operations or independent development of functionally similar clones. Different textual, syntactic, and semantic proximity measures can be used to determine software similarities such as similarity of code, abstract syntax tree, or program control flow. This early work, we consider CWE vulnerability type, source code, and project category pertaining to the logic, to capture projects similarity.

Vulnerability. We focus on C/C++ code vulnerabilities reported by National Vulnerability Database (NVD) with tagged Common Vulnerabilities and Exposures (CVEs) that is a list of common identifiers for publicly known security vulnerabilities, the affected project versions, and Common Weakness Enumerations (CWEs) [1] that is a list of common software and hardware weaknesses. With regards to the definition of *"the same vulnerability"*, in this work we assume that projects which their vulnerabilities belong to *"the same CWE category"* have the same vulnerability. CWE list is supported by an active

TABLE I: Our C/C++ subject systems.

ID	Category	# Subject systems	# Vulnerable functions	# CVEs	# CWEs
C1	Networking, Net Analysis/Monitoring, and Protocols	29	403	181	22
C2	Security, Cryptography, Forensics, Cryptocurrency	27	437	176	31
C3	Programming Libraries and SDKs	22	490	150	31
C4	Operating Systems and OS Utilities	22	3,393	1,121	56
C5	IRC Client/Server, Email, and Messaging Tools and Libraries	20	80	39	17
C6	Image Viewing, Manipulation, and Processing	20	654	265	28
C7	Software Development Tools and Libraries	15	152	73	16
C8	Parsers, Editors, and Text Processors	14	80	27	15
C9	Remote Access, VPN, FTP, SSH	13	98	33	15
C10	Multimedia Tools, Streaming, Frameworks, and Libraries	14	219	128	22
C11	Windowing Systems, Display Servers, and Libraries	11	56	31	14
C12	System and Service Managers	11	112	28	15
C13	Document Viewers, Analyzers, Fonts, and Converters	11	171	81	19
C14	Hardware/Middleware/Firmware Tools and Libraries	10	75	19	12
C15	File System Software and Utilities	10	36	15	10
C16	Compression/Archiver Tools and Libraries	9	54	42	16
C17	Authentication Tools and Libraries	8	12	9	4
C18	Web Servers	7	27	15	7
C19	Web Browsers/Clients	6	3,953	664	39
C20	Proxy Servers and Overlay Network	6	17	10	8
C21	Others	6	20	10	8
C22	Database Tools and DBMS	6	60	14	10
C23	Interpreters, Compilers, Disassemblers	5	102	60	9
C24	Games, Game Engines, and Simulators	5	27	5	5
C25	Virtualization	4	142	84	21
C26	Monitoring, Data Collection, Logging, and Reporting	4	10	6	6
	Total	315	10,880	3,284	89

community and its list of weakness types generally serves as a common baseline for weakness identification that is adequately described and differentiated [1].

Note that even if a *vulnerable* function is used safely in the code, i.e. benign usages, we still consider that vulnerability at the function level.

Granularity. In our early experiments we found that files are too coarse a granularity for code similarity comparison, and file-based similarity analysis is not suitable for similar code vulnerability prediction. Hence in this paper, we consider function-level similarity analysis as a more valid approach.

B. Data Collection and Subject Systems

We use a public dataset[1] called Big-Vul [9] that is a large collection of C/C++ code vulnerability from open-source Github projects. The authors crawled the public CVE database and CVE-related source code repositories and extracted vulnerability-related code changes. Almost every CVE has the *Weakness Enumeration* field, which contains at least one CWE. Each record in the Big-vul dataset has a security characteristics (CWE) ID that caused the vulnerability. Big-Vul contains 3,754 C/C++ code vulnerabilities (CVEs), spanning 91 different vulnerability types (CWEs) extracted from 348 Github projects. This dataset links project repositories on Github with CVE database, project bug reports, and code commits. Also, Big-Vul annotates the source code of projects and their relevant code commits to extract vulnerable functions.

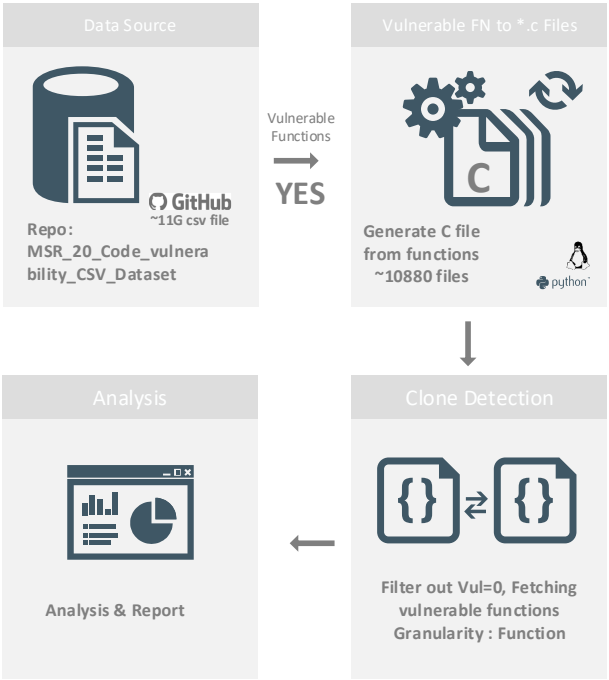


Fig. 1: Data collection process.

Figure 1 depicts our data collection process. We wrote a script using Pandas[2] to parse the data from Big-Vul that contains projects name, details of CVEs, CWE IDs and

[1]https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

[2]https://pandas.pydata.org/

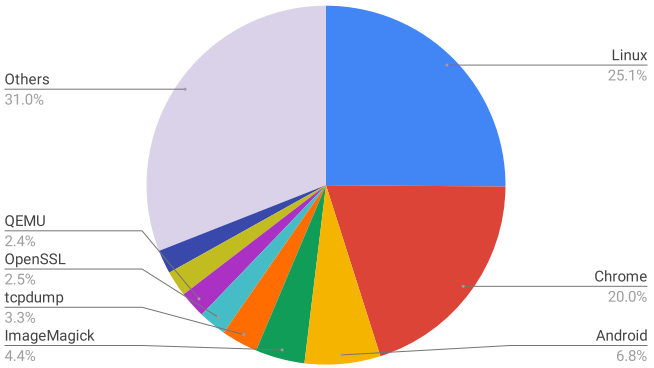


Fig. 2: Distribution of C/C++ projects to CVE count.

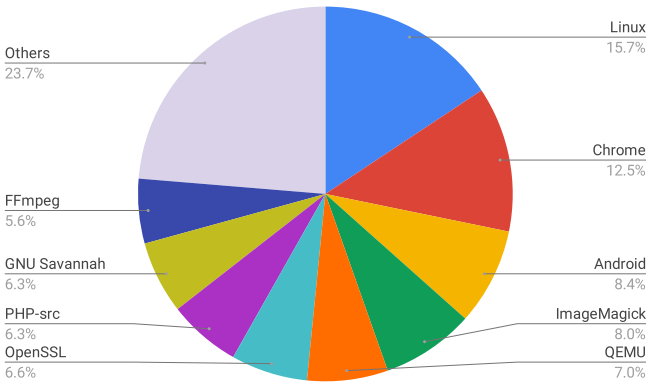


Fig. 3: Distribution of C/C++ projects to CWE count.

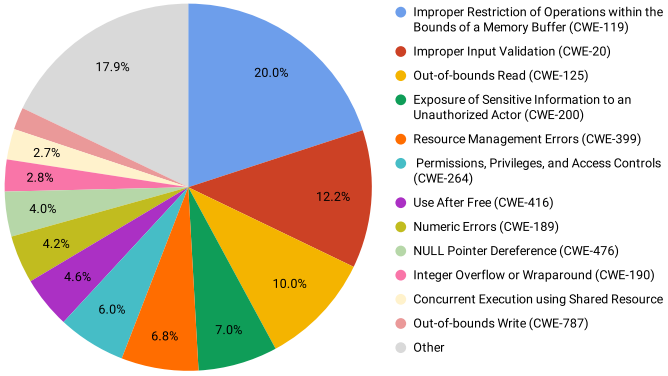


Fig. 4: Distribution of CWE count.

vulnerable functions for each project. Big-Vul dataset contains both positive and negative samples for vulnerable functions. Thus we filter out and kept only positive samples to analyze the similarity across different vulnerable functions. In total, we extracted 89 different CWEs out of 3,284 collected CVEs.

After fetching raw data from their CSV file, we generate separate C/C++ code files each containing a vulnerable function for code similarity comparison across the functions. This is because the code similarity tool that we use gets as input a set of files and since our granularity is at function-level, we generate separate C/C++ files to analyze the similarity of functions.

In order to determine project categories, we used Github description, Wikipedia type information, and the corresponding project websites. After carefully assigning project categories we ended up with 26 categories that are shown in Table I.

Figure 2 present the overall distribution of projects to CVE count where Linux and Chrome have the highest number of CVEs with 825 (25%) and 656 (20%) counts respectively. Figure 3 present the overall distribution of projects to CWE count. Again Linux and Chrome have the highest number of CWEs with 45 (15.7%) and 36 (12.5%) counts respectively. Figure 4 shows the distribution of the CWE counts where Improper Restriction of Operations within the Bounds of a Memory Buffer (`CWE-119`), Improper Input Validation (`CWE-20`) and Out-of-bounds Read (`CWE-125`), have the highest number of CVEs with 591, 360, and 295 restrictively.

C. Similarity Analysis

In RQ1 (vulnerabilities in similar code) we would like to see how similar are code snippets (functions) with the same vulnerability. To answer RQ1 we calculate the code similarity of the vulnerable functions that is specified in the Big-Vul dataset. We first tested two clone detectors: SourcererCC [26] and Cloneworks [28]. However, they did not report the exact value of similarity for each clone pair. SourcererCC gives us an overall amount of similarity in the whole dataset and CloneWorks indicates a clone pair without reporting the exact similarity value. Therefore for this initial study we used JPlag [4] that is a tool for finding similarities between multiple sets of source code. It generates token strings from the source code and computes a percentage score. JPlag compares programs by the amount that sequences from one covers the other, and the program coverage is defined as the ratio of covered code to the total code. We discuss the possibility of using other code similarity measurement tools in the threats to validity part in Section III-D.

The domain or category of a software project can pertain to its program logic. We answer RQ2 (vulnerabilities in the same domain) by computing the percentage of projects with the same category that have the same most occurring CWEs. This is described more in Section III-B.

III. Results

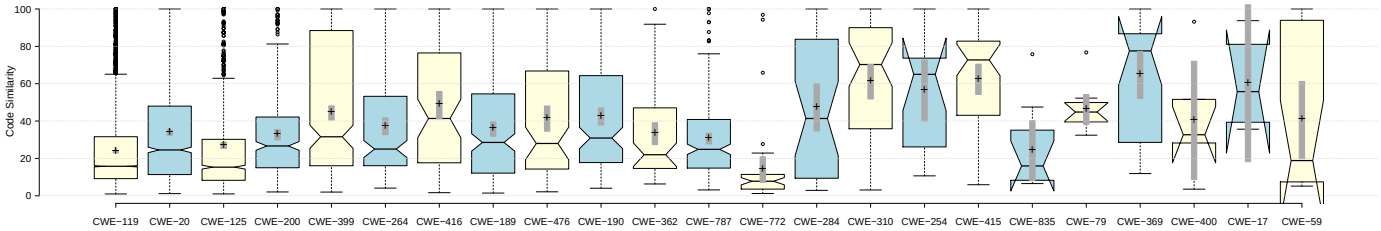
A. Vulnerabilities in Similar Code (RQ1)

The software engineering community does not have a consensus on what exactly a code clone is, and there is no specific minimum threshold for token-based clone detection [11]. For our purpose, we consider the similarity of around 50% and above as a basis to determine two code to be similar.

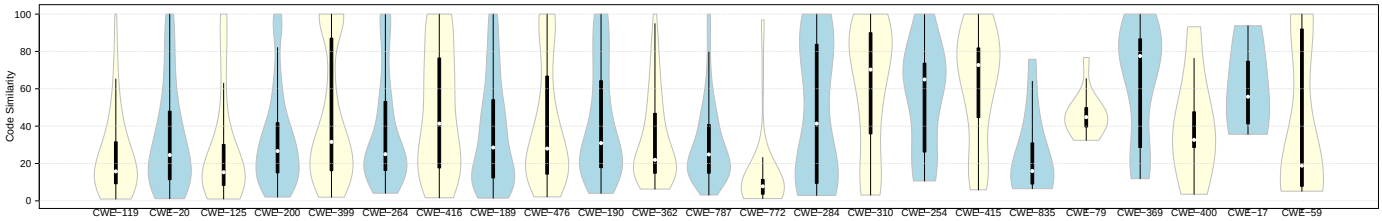
In our dataset we have 89 different CWEs associated with 3,284 CVEs and each CWE has a different number of CVEs. In order to select CWEs with enough CVEs to be representative for our study, we only select CWEs which are

TABLE II: Statistics of the most occurring CWEs that are found in more than 15 different CVEs.

CWE ID	CWE Category	CVEs Count	Vulnerable Functions	Top 40 CWE Ranking 2019	2020	2021
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	591	2124	1	5	17
CWE-20	Improper Input Validation	360	1141	3	3	4
CWE-125	Out-of-bounds Read	295	620	5	4	3
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	207	503	4	7	20
CWE-399	Resource Management Errors	201	736	–	–	–
CWE-264	Permissions, Privileges, and Access Controls	177	508	–	–	–
CWE-416	Use After Free	135	328	7	8	7
CWE-189	Numeric Errors	125	337	–	–	–
CWE-476	NULL Pointer Dereference	117	215	14	13	15
CWE-190	Integer Overflow or Wraparound	83	307	8	11	12
CWE-362	Concurrent Execution using Shared Resource with Improper Sync. ('Race Condition')	80	277	29	34	33
CWE-787	Out-of-bounds Write	58	198	12	2	1
CWE-772	Missing Release of Resource after Effective Lifetime	38	46	21	–	–
CWE-284	Improper Access Control	38	176	–	30	–
CWE-310	Cryptographic Issues	31	94	–	–	–
CWE-254	7PK - Security Features	29	122	–	–	–
CWE-415	Double Free	28	81	31	38	–
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	23	34	26	36	–
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	23	55	2	1	2
CWE-369	Divide By Zero	22	34	–	–	–
CWE-400	Uncontrolled Resource Consumption	21	48	20	23	27
CWE-17	Code Development (Specification, Design, and Implementation)	18	51	–	–	–
CWE-59	Improper Link Resolution Before File Access ('Link Following')	15	52	–	40	31



(a) Notched boxplots of code similarities. Mean values are shown with + and 95% confidence interval of means in gray box.



(b) Violin plots of code similarities.

Fig. 5: Code similarity comparison within each CWE category.

found in at least 15 different CVEs and therefore, we ended up with 23 CWEs as shown in Table II. For each CWE, we then calculate the similarity of pair-wise vulnerable functions that we stored in separate files. We also present in Table II the CWE Top Most Dangerous Software Weaknesses for 2019, 2020, and 2021 (CWE Top 25)[3], which are the most common and impactful issues experienced over the previous two calendar years.

Figure 5 depicts the statistics for code similarity within each CWE. We observe that only 8 out of 23 CWEs (35%), including CWE-369 (Divide By Zero), CWE-415 (Double Free), CWE-310 (Cryptographic Issues), CWE-254 (7PK - Security Features), CWE-17 (Code Development), CWE-79 (Cross-site Scripting), CWE-284 (Improper Access Control), and CWE-416 (Use After Free) are found in similar code, i.e., have relatively high (mean and median of about 50% and above) code similarity. Considering the means, medians, and distributions, the code similarity between vulnerable functions within most CWEs categories is not generally high. This suggests that the codes for different CVE with the same CWE are not much similar in general but in those 8 CWEs.

We observed two interesting results regarding CWE-772

[3]https://cwe.mitre.org/top25/

TABLE III: Statistics for the average similarities and the most occurring CWEs within each project category.

ID	Category	Avg of Code Similarities	Median of Code Similarities	Most Occurring CWE	Most Occurring CWE Count	Num of Projects	Ratio of Most Occ CWE Category
C1	Networking, Net Analysis/Monitoring, and Protocols	24.272	17.284	CWE-119	14	29	0.483
				CWE-20	9	29	0.310
C2	Security, Cryptography, Forensics, Cryptocurrency	26.406	18.182	CWE-119	12	27	0.444
				CWE-20	9	27	0.333
C3	Programming Libraries and SDKs	22.670	14.655	CWE-119	10	22	0.455
C4	**Operating Systems and OS Utilities**	26.430	21.818	**CWE-119**	12	22	**0.545**
C5	IRC Client/Server, Email, and Messaging Tools and Libraries	20.784	13.393	CWE-20	6	20	0.3
C6	**Image Viewing, Manipulation, and Processing**	18.890	12.632	**CWE-119**	10	20	**0.5**
				CWE-125	8	20	0.4
				CWE-190	7	20	0.35
				CWE-369	7	20	0.35
C7	Software Development Tools and Libraries	20.866	15.789	CWE-119	6	15	0.4
C8	Parsers, Editors, and Text Processors	21.396	12.245	CWE-119	5	14	0.357
C9	Remote Access, VPN, FTP, SSH	25.484	23.529	CWE-119	6	13	0.462
C10	**Multimedia Tools, Streaming, Frameworks, and Libraries**	22.612	14.773	**CWE-119**	11	14	**0.786**
				CWE-125	7	14	**0.5**
C11	Windowing Systems, Display Servers, and Libraries	22.455	19.403	CWE-190	3	11	0.273
C12	System and Service Managers	26.824	24.242	CWE-119	4	11	0.364
C13	Document Viewers, Analyzers, Fonts, and Converters	19.278	13.542	CWE-119	5	11	0.455
				CWE-125	5	11	0.455
				CWE-476	4	11	0.364
C14	Hardware/Middleware/Firmware Tools and Libraries	26.085	22.222	CWE-119	3	10	0.3
C15	File System Software and Utilities	21.308	14.607	CWE-119	3	10	0.3
C16	Compression/Archiver Tools and Libraries	18.231	12	CWE-119	3	9	0.333
				CWE-189	3	9	0.333
				CWE-476	3	9	0.333
				CWE-22	3	9	0.333
C17	Authentication Tools and Libraries	21.038	17.391	CWE-119	3	8	0.375
C18	Web Servers	27.881	17.978	CWE-264	2	7	0.286
				CWE-20	2	7	0.286
C19	Web Browsers/Clients	39.445	34.286	CWE-264	2	6	0.333
				CWE-119	2	6	0.333
				CWE-20	2	6	0.333
				CWE-416	2	6	0.333
				CWE-476	2	6	0.333
C20	Proxy Servers and Overlay Network	20.172	13.684	CWE-119	2	6	0.333
C21	Others	18.528	14.130	CWE-125	2	6	0.333
C22	**Database Tools and DBMS**	30.083	24.528	**CWE-119**	4	6	**0.667**
C23	**Interpreters, Compilers, Disassemblers**	19.475	12.697	**CWE-125**	4	5	**0.8**
				CWE-119	3	5	**0.6**
				CWE-190	3	5	**0.6**
				CWE-476	3	5	**0.6**
				CWE-416	2	5	0.4
				CWE-787	2	5	0.4
C24	Games, Game Engines, and Simulators	39.598	36.364	-	-	5	-
C25	**Virtualization**	28.929	22.642	**CWE-20**	2	4	**0.5**
				CWE-617	2	4	**0.5**
C26	Monitoring, Data Collection, Logging, and Reporting	14.030	9.060	-	-	4	-

and CWE-79. We notice that CWE-772 has the least pair-wise similarity between functions in the same category. According to the 2020 CWE Top 25 [4], in 2019 CWE-772 (Missing Release of Resource after Effective Lifetime) was in 21st place; however, it does not explain which type of resource not being released was important, and therefore in the 2020 list, a more specific mapping was used to show the exact type of resource. Due to this change, CWE-401 (Missing Release of Memory after Effective Lifetime) went from not being on the top 50 to being 32nd, and CWE-772 representing all non-memory resources dropped to 75th. Our result is also in line with their decision as functions with CWE-772 are of diverse nature. On the other hand, CWE-79 has about 50% mean and median similarity as well as higher pair-wise minimum similarity compared to the rest of CWEs. This is a good sign for better accuracy of code analyzer tools that target detecting such vulnerabilities since CWE-79 is at the top of the CWE Top Ranking list.

[4]https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

Finding 1: Code with the same vulnerabilities are not very similar in general. Only functions with vulnerabilities in CWE-369 (Divide By Zero), CWE-415 (Double Free), CWE-310 (Cryptographic Issues), CWE-254 (7PK - Security Features), CWE-17 (Code Development), CWE-79 (Cross-site Scripting), CWE-284 (Improper Access Control), and CWE-416 (Use After Free), have mean and median above our minimum threshold of 50% code similarity.

B. Vulnerabilities in the Same Domain/Category (RQ2)

Table III presents code similarity of vulnerable functions in each project category. It is expected that we do not see high similarity because there are different types of functions in each project in that category. However, interestingly for two categories of *Web Browser/Clients* and *Games, Game Engines, and Simulators*, we observe relatively high mean and median values for code similarity between vulnerable functions.

To investigate RQ2, we count every occurrence of CWEs for projects in each category, list the most occurring ones, and compute the ratio of projects with a vulnerability in that CWE with respect to the total number of projects in that category. We consider a CWE to be frequent in a category if it occurs in at least 50% of the projects in that category. It is evident from Table III that the ratio of vulnerabilities CWE-119, CWE-125, CWE-190, CWE-476, CWE-20, and CWE-617 is at least 50% only in categories C4, C6, C10, C22, C23, and C25 (6/26 = 23%). Note that CWE-119, CWE-20, and CWE-125 are already frequent as shown in Table II.

Finding 2: Same vulnerabilities are more frequent within specific categories of projects. CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) is frequent in *OS and OS Utilities*, *Image Viewing, Manipulation, and Processing*, *Multimedia Tools, Streaming, Frameworks, and Libraries*, *DB Tools and DBMS*, and *Interpreters, Compilers, Disassemblers*. CWE-125 (Out-of-bounds Read) is frequent in *Multimedia Tools, Streaming, Frameworks, and Libraries*, and *Interpreters, Compilers, Disassemblers*. CWE-190 (Integer Overflow or Wraparound) and CWE-476 (NULL Pointer Dereference) are frequent in *Interpreters, Compilers, Disassemblers*. CWE-20 (Improper Input Validation) and CWE-617 (Reachable Assertion) are frequent in *Virtualization*.

C. Prevalence of Vulnerabilities (RQ3)

Table II lists the most prevalent CWEs in our study sorted by their number of instances. In RQ3, we are interested to know which of these CWEs are the most repeating ones within similar codes. The results shown in Figure 5 suggest that CWE-416 (Use After Free) is the most prevalent vulnerability in similar code, which is at 7th position among the overall most prevalent ones as shown in Table II. Other prevalent CWEs with high code similarity are CWE-284, CWE-310, CWE-254, CWE-415, CWE-79, CWE-369, and CWE-17, which all are at the lower half of table II.

Finding 3: The most prevalent vulnerabilities that occur in similar code are (Use After Free), CWE-284 (Improper Access Control), CWE-310 (Cryptographic Issues), CWE-254 (7PK - Security Features), CWE-415 (Double Free), CWE-79 (Cross-site Scripting), CWE-369 (Divide By Zero), and CWE-17 (Code Development). However, these CWEs are less frequent in total across all subjects.

D. Discussion

Findings 1 and 2 indicate that vulnerable functions in 35% (8 out of 23) of the most occurring CWEs have similar code, and 23% (6 out of 26) of projects with same domain/category have same vulnerabilities. Overall, the code similarity among functions with the same CWE is not very high. The results might, however, be different if we use other code similarity calculator tools. Also, some researchers [13], [15], [16] used patch or code block level granularity and reported high similarities. This could be a future enhancement to our work. This study needs to be extended by more fine-grained experiments. Currently if a CVE patch changes one line of a large function, that function is used for similarity comparison. Moreover, we defined the same vulnerabilities as those that are in the same CWE category which can also be coarse-grained. For instance there can be different input validation problems in the same category. Thus we may need to consider more semantic features of CVEs.

Implications. While more analysis is needed, current findings give insight into what security issues propagate due to code reuse. Our results suggest that vulnerabilities that occur more in clone or near-duplicate code are:

- CWE-416: Use After Free
- CWE-284: Improper Access Control
- CWE-310: Cryptographic Issues
- CWE-254: 7PK - Security Features
- CWE-415: Double Free
- CWE-79: Cross-site Scripting
- CWE-369: Divide By Zero

One of the goals of this research as mentioned in the introduction is to investigate whether existing vulnerabilities of an application can be used to locate the same vulnerabilities in applications that share similar code. Researchers who wish to develop an automated vulnerability detection tool based on code similarity or abstract patterns, can tailor their algorithms more towards the above-mentioned security flaws. Also, as shown by Gkortzis et al. [10] lack of test code is a reason for undetected vulnerabilities. In this regard, previous research suggest that existing test code can be used to generate new tests for similar applications [19], [20]. It would be interesting to see if security tests can be transplanted as well.

Also, knowing which CWEs are more occurring in particular types of applications (Table III) can be helpful both for researchers and developers. According to our results, less frequent CWEs across all projects but dominant in special project categories are CWE-476 and CWE-190 in C23 (Interpreters, Compilers, Disassemblers), and CWE-617 in C25

(virtualization). More frequent CWEs that are dominant in special project categories are CWE-119 in {C4, C6, C10, C22, and C23}, CWE-20 in C25 (virtualization), and CWE-125 in C10 and C23.

Threats to validity. An external validity threat is that our definition for "same vulnerability" and thus our results and findings are based on the CWE databases. We acknowledge that this might not always be correct as for instance, there can be different input validation problems in the same CWE category of missing input validation. However, since it is well-maintained by the authorities and is used in several other research work, we believe it is representative. Another external validity threat is regarding the generalization of our results. We acknowledge that more projects and different languages should be evaluated. However, we believe that the studied subjects represent real-world projects that differ in category, size, maturity, etc. Also, our thresholds for determining similar code as well as using other code similarity calculator tools can affect our findings. Using tools such as GumTree [3] that considers Abstract Syntax Tree can provide better insights; however, it is more a visualized diff and needs manual inspection. An internal validity threat is that the project category was defined by us, which may introduce author bias. To mitigate this, we used Github description, Wikipedia type information, and the corresponding project websites.

IV. Related Work

An et al. [5] examined C code vulnerabilities by similarity matching between the security characteristics and the code. Gkortzis et al. [10] studied software vulnerabilities and showed projects without test code or continuous integration suffer from undetected vulnerabilities. Livshits et al. [17] analyzed common JAVA vulnerabilities. Li et al. [13] applied static and dynamic analysis to discover code clone vulnerability based on released security patches. Perrand et al. investigated JAVA component vulnerabilities [23]. Neuhaus et al. [22] studied the Mozilla vulnerability history and found that components with similar imports or function calls were likely to be vulnerable. Similar work by Scandariato et al. [27] analyzed vulnerable components in Android apps. Du et. al [8] proposed and implemented a generic framework to identify potentially vulnerable functions through program metrics. However, their approach was not designed to directly pinpoint vulnerabilities but to assist confirmative vulnerability assessment. Ponta et. al [24] manually collected and curated a dataset of vulnerabilities of open-source Java software and their fixing commits. They used their dataset to train classifiers that could automatically identify security-relevant commits in code repositories.

A few work studied code clone vulnerabilities [7], [12], [13], [15], [16], [31]. Dang et al. [7] explored code-clone detection in industrial practices and used the tool the Microsoft Security Response Center to investigate security vulnerabilities. Li et al. [13] propose a mix of static and dynamic analysis to discover code clone vulnerability based on released security patches. Kim et al. [12] present an approach for code clone detection at functions level and extracted a pool of vulnerability patches from CVE. Li et al. [15] performed a similar approach that automatically selects the code-similarity algorithm that is effective for a specific vulnerability. Liu et al. [16] proposed a system that can detect vulnerable code clones based on fingerprints. Verdi et al. [31] investigated security vulnerabilities in C++ code snippets on Stack Overflow and developed a browser extension to check for vulnerabilities in code snippets when they upload them on the platform.

V. Conclusions and Future Work

We studied vulnerabilities in C/C++ code to investigate the possibility of predicting security issues within similar applications. Our results showed that the average code similarity of vulnerable functions with the same CWE is not very high in general. The most prevalent vulnerabilities that occur in similar code are *Use After Free*, *Improper Access Control*, *Cryptographic Issues*, *7PK - Security Features*, *Double Free*, *Cross-site Scripting*, and *Divide By Zero*, which are less frequent compared to other CWEs across all subjects. Researchers who wish to develop an automated vulnerability detection tool based on code similarity or abstract patterns can tailor their algorithms to such security flaws. We plan to use other similarity calculator tools and conduct a similar study for other languages such as Java, Python, and JavaScript. Another interesting research direction is to evaluate security flaws that occurred in the test clone [29] compared to the production code clone [30].

References

[1] CWE. http://cwe.mitre.org/data/index.html, 2021.

[2] GitHub Advisory Database. https://github.com/advisories, 2021.

[3] GumTree. https://github.com/GumTreeDiff/gumtree, 2021.

[4] JPlag. https://github.com/jplag/jplag, 2021.

[5] X. An, W. Li, and W. Pan. Code based software security vulnerability analyzing and detecting based on similar characteristic. In *International Conference on Intelligent System and Knowledge Engineering*, 2008.

[6] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder. Effects of cloned code on software maintainability: A replicated developer study. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013.

[7] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie. Transferring code-clone detection and analysis to practice. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017.

[8] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE, 2019.

[9] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[10] A. Gkortzis, D. Mitropoulos, and D. Spinellis. Vulinoss: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2018.

[11] Y. Golubev, M. Eliseeva, N. Povarov, and T. Bryksin. A study of potential code borrowing and license violations in java projects on github. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 2020.

[12] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017.

[13] H. Li, H. Kwon, J. Kwon, and H. Lee. A scalable approach for vulnerability discovery based on security patches. In *Applications and Techniques in Information Security*. Springer Berlin Heidelberg, 2014.

[14] H. Li, H. Kwon, J. Kwon, and H. Lee. CLORIFI: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, 28(6):1900–1917, 2015.

[15] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Annual Conference on Computer Security Applications*. ACM, 2016.

[16] Z. Liu, Q. Wei, and Y. Cao. VFDETECT: A vulnerable code clone detection system based on vulnerability fingerprint. In *IEEE Information Technology and Mechatronics Engineering Conference*. IEEE, 2017.

[17] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[18] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[19] A. Milani Fard. *Directed test generation and analysis for web applications*. PhD thesis, University of British Columbia, 2017.

[20] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 67–78. ACM, 2014.

[21] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. Investigating context adaptation bugs in code clones. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.

[22] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the ACM Conference on Computer and communications security (CCS)*. ACM, 2007.

[23] P. Parrend. Enhancing automated detection of vulnerabilities in java components. In *2009 International Conference on Availability, Reliability and Security*. IEEE, 2009.

[24] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *International Conference on Mining Software Repositories (MSR)*. IEEE, 2019.

[25] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Working Conference on Reverse Engineering (WCRE)*, pages 81–90. IEEE, 2008.

[26] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[27] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[28] J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with cloneworks. In S. Uchitel, A. Orso, and M. P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering (ICSE) - Companion Volume*, pages 27–30. IEEE Computer Society, 2017.

[29] A. Vahabzadeh, A. Milani Fard, and A. Mesbah. An empirical study of bugs in test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE Computer Society, 2015.

[30] B. van Bladel and S. Demeyer. Clone detection in test code: an empirical evaluation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 492–500. IEEE, 2020.

[31] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *arXiv preprint arXiv:1910.01321*, 2019.