Security Vulnerabilities in Categories of Clones and Non-Cloned Code: An Empirical Study

Md Rakibul Islam
University of New Orleans, USA
Email: mislam3@uno.edu

Minhaz F. Zibran
University of New Orleans, USA
Email: zibran@cs.uno.edu

Aayush Nagpal
University of New Orleans, USA
Email: anagpal@uno.edu

Abstract—**Background: Software security has drawn immense importance in the recent years. While efforts are expected in minimizing security vulnerabilities in source code, the developers' practice of code cloning often causes multiplication of such vulnerabilities and program faults. Although previous studies examined the bug-proneness, stability, and changeability of clones against non-cloned code, the security aspects remained ignored. Aims: The objective of this work is to explore and understand the security vulnerabilities and their severity in different types of clones compared to non-clone code. Method: Using a state-of-the-art clone detector and two reputed security vulnerability detection tools, we detect clones and vulnerabilities in 8.7 million lines of code over 34 software systems. We perform a comparative study of the vulnerabilities identified in different types of clones and non-cloned code. The results are derived based on quantitative analyses with statistical significance. Results: Our study reveals that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. However, the proportion (i.e., density) of vulnerabilities in clones and non-cloned code does not have any significant difference. Conclusion: The findings from this work add to our understanding of the characteristics and impacts of clones, which will be useful in clone-aware software development with improved software security.**

I. Introduction

Software security has become one of the most pressing concerns recently. Software developers are expected to write secure source code and minimize security vulnerabilities in the systems under development. However, the developers' copy-paste practice for code reuse often cause the multiplication and propagation of program faults and security vulnerabilities [22], [25], [52]. Thus, there is a possibility that such vulnerabilities can exist in cloned code at a higher rate.

Code clone (i.e., similar or duplicated code) is already identified as a notorious code smell (i.e., a symptom indicating source of future problems) [14] that cause other problems such as reduced code quality, code inflation, and change difficulties [25], [31], [50], [52]. Nevertheless, software systems typically have 9%-17% [53] cloned code, and the proportion is sometimes found to be even 50% [37] or higher [13].

Clones are arguably a major contributor to the high maintenance cost for software systems, and as much as 80% of software costs are spent on maintenance [20]. Thus, to understand the characteristics and contexts of the detrimental impacts of clones, earlier studies examined the comparative stability of clones as opposed to non-cloned code [16], [18], [27], [28], [34], relationships of clones with bug-fixing changes [11], [19], [24], [25], [26], [30], [36], [43], [49], the impacts of clones on program's changeability [17], [31], [32] and comparative fault-proneness of cloned and non-cloned code [22], [42].

However, the security aspects of code clones have never been studied before, although the reuse of vulnerable components and source code (i.e., code clones) multiply security vulnerabilities [23], [29], [46]. This paper presents, a large empirical study on the security vulnerabilities in code clones and presents a comparative analysis of these vulnerabilities in different types of clones as opposed to non-cloned code. In particular, we address the following five research questions.

RQ1: *Do code clones contain higher number of security vulnerabilities than non-cloned code or vice versa*?
— The answer to this question will add to our understanding of the negative impacts of clones, which will be useful in *cost-benefit analysis* [40] for improved clone management.

RQ2: *Do clones of a certain category contain more security vulnerabilities than others?*
— If a certain type of clones are found to have higher number of security vulnerabilities, those clones will be high-priority candidates for removal or careful maintenance.

RQ3: *Do code clones contain more severe (i.e. riskier) vulnerabilities compared to non-cloned code or vice versa*?
— All the security vulnerabilities are not equally risky in terms of security threat. The result of this research question will advance our understanding of clones' impacts and will be useful in cost-effective clone management [40].

RQ4: *Do clones of a certain category contain relatively severe (i.e., riskier) security vulnerabilities than others?*
— If a certain type of clones are found to have riskier security vulnerabilities, those clones will demand especial attention and high-priority in clone-removal process.

RQ5: *Can we distinguish some security vulnerabilities that appear more frequently in cloned code as opposed to non-cloned code?*
— If we can distinguish a set of vulnerabilities that appear more frequently in code clones, the finding will help software developers to stay cautious of such vulnerabilities while cloning source code. In addition, that particular set of vulnerabilities can be minimized by clone refactoring.

To answer the aforementioned research questions, we conduct a large-scale empirical study over 8.7 million lines of

978-1-5090-4039-1/17 $31.00 © 2017 IEEE
DOI 10.1109/ESEM.2017.9

source code in 34 open-source software systems written in C. Using a wide range of metrics and characterization criteria, we carry out in-depth quantitative analyses on the source code of the systems with respect to different categories of code clones, non-cloned code, and a set of security vulnerabilities. In this regard, this paper makes the following two contributions:

- We present a large-scale comparative study of the security vulnerabilities in code clones and non-cloned code. To the best of our knowledge, no such study of the security vulnerabilities in code clones exists in the literature.
- We also perform a comparative analysis of security vulnerabilities in different types of clones (e.g., *Type-1*, *Type-2*, and *Type-3*), which informs the relative security implications of clones at different similarity levels.

II. Terminology and Metrics

In this section, we describe and define the terminologies and metrics that are used in our work. Some of the metrics are adapted from the literature [22], [38].

A. Security Vulnerabilities

A software security vulnerability is defined as a weakness in a software system that can lead to a compromise in integrity, availability or confidentiality of that software system. For example, *buffer overflow* and *dangling pointers* are two well known security vulnerabilities. The cyber security community maintains a community-developed list of common software security vulnerabilities where each category of vulnerability is enumerated with a CWE (Common Weakness Enumeration) number [1]. For example, CWE-120 refers to those vulnerabilities that fall into the CWE category of *classic buffer overflow*. More examples of security vulnerabilities along with their CWE enumerations are presented in Table I.

B. Code Clones

Our study includes code clones at the granularity of syntactic blocks (located between curly braces { and }) known as *block clones*. We study clones at different levels of syntactic similarities as mentioned below.

TABLE I
Security vulnerabilities frequently identified in the systems

Security Vulnerability	Description
Buffer Overflow	An application attempts to write data past the end of a buffer (CWE-120).
Uncontrolled Format String	Submitted data of an input string is evaluated as a command by the application (CWE-134).
Integer Overflow	The result of an arithmetic operation exceeds the maximum size of the integer type used to store it (CWE-190).
Null Pointer Dereference	Dereference a pointer that is null (CWE-476).
Memory Leak	Not release allocated memory (CWE-401).
Null Termination Errors	A string is incorrectly terminated (CWE-170).
Concurrency Errors	Concurrent execution using shared resource with improper synchronization (CWE-362).
Double Free	The program calls free() twice on the same memory address (CWE-415).
Access Freed Memory	Access memory after it has been freed (CWE-416).
Insecure Randomness	The software may use insufficiently random numbers or values in a security context that depends on unpredictable numbers (CWE-330).
OS Command Injection	Improper neutralization of special elements used in an operating systems command (CWE-78).
Insecure Temporary File	Creating and using insecure temporary files can leave application and system data vulnerable to attack (CWE-377).
Reliance on Untrusted Inputs	The input can be modified by an untrusted actor in a way that bypasses the protection mechanism (CWE-807).
Poor Code Quality	Indication that the product has not been carefully developed or maintained (CWE-398).
Dead Code	Code that can never be executed (CWE-561).
Use of Obsolete Functions	The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained (CWE-477).
Risky Cryptography	Stealing the information protected, under normal conditions, by the SSL/TLS encryption (CWE-327).
Unused Variable	The variable's value is assigned but never used (CWE-563).

Type-1 Clones: Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called *Type-1* clones [51].

Type-2 Clones: *Type-2* clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout, and comments [51].

Type-3 Clones: Code fragments, which exhibit similarities as of *Type-2* clones and also allow further differences such as additions, deletions or modifications of statements are known as *Type-3* clones [51].

By the definitions above, *Type-2* clones include *Type-1* while *Type-3* clones include both *Type-1* and *Type-2*. Let, T_1, T_2, and T_3 respectively denote the sets of *Type-1*, *Type-2*, and *Type-3* clones in a software system. Mathematically, $T_1 \subseteq T_2 \subseteq T_3$. Thus, two additional subsets of *Type-2* and *Type-3* clones are characterized as follows.

Pure Type-2 Clones: A set of pure *Type-2* clones include only those *Type-2* clones that do not exhibit *Type-1* similarity. Mathematically, $T_2^p = T_2 - T_1$, where T_2^p denotes the set of pure *Type-2* clones.

Pure Type-3 Clones: A set of pure *Type-3* clones include only those *Type-3* clones, which do not exhibit similarities at the levels of *Type-1* or *Type-2* clones. Mathematically, $T_3^p = T_3 - T_2$, where T_3^p denotes the set of pure *Type-3* clones.

C. Metrics

The required metrics are defined in terms of density of vulnerabilities with respect to (w.r.t.) syntactic blocks of code (BOC) as well as w.r.t. lines of code (LOC). Only source lines of code are taken into consideration excluding comments and blank lines.

Let, $\mathcal{C}$ denote the set of all *cloned* code blocks and $\bar{\mathcal{C}}$ denote the set of all *non-cloned* code blocks.

Also let, $\mathcal{V}_{\mathcal{C}}$ denote the set of vulnerabilities found in $\mathcal{C}$ and $\mathcal{V}_{\bar{\mathcal{C}}}$ denote the set of vulnerabilities located in $\bar{\mathcal{C}}$. A cloned code block in $\mathcal{C}$ can be of category $\mathcal{X}$ clone where $\mathcal{X} \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$. Thus, $\mathcal{V}_{\mathcal{C}}$ can be split into multiple sets with $\mathcal{V}_{\mathcal{X}}$ denoting the set of vulnerabilities identified in clones of category $\mathcal{X}$.

Density of vulnerabilities w.r.t. BOC in category $\mathcal{X}$ clones, denoted as $\partial_{\mathcal{X}}^{\beta}$, is defined as the ratio of the number of vulnerabilities found in clones of category $\mathcal{X}$ to the number of cloned blocks in category $\mathcal{X}$ clones. Mathematically,

$$\partial_{\mathcal{X}}^{\beta} = \frac{|\mathcal{V}_{\mathcal{X}}|}{\beta_{\mathcal{X}}} \tag{1}$$

where $|\mathcal{V}_{\mathcal{X}}|$ denotes the number of vulnerabilities found in the blocks of category $\mathcal{X}$ clones and $\beta_{\mathcal{X}}$ denotes the total number of block clones of category $\mathcal{X}$. And, $\mathcal{X} \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$.

Density of vulnerabilities w.r.t. BOC in type $\mathcal{T}$ code, denoted as $\partial_{\mathcal{T}}^{\beta}$, is defined as the ratio of the number of vulnerabilities found in type $\mathcal{T}$ code to total number of blocks in type $\mathcal{T}$ code, where $\mathcal{T} \in \{\mathcal{C}, \bar{\mathcal{C}}\}$. Mathematically,

$$\partial_{\mathcal{T}}^{\beta} = \frac{|\mathcal{V}_{\mathcal{T}}|}{\beta_{\mathcal{T}}} \tag{2}$$

where $\mathcal{V}_\mathcal{T} \in \{\mathcal{V}_\mathcal{C}, \mathcal{V}_{\bar{\mathcal{C}}}\}$ and $\beta_\mathcal{T}$ denotes the total number of blocks in type $\mathcal{T}$ code.

Density of vulnerabilities per 1,000 LOC (KLOC) in category $\mathcal{X}$ clones, denoted as $\partial^\ell_\mathcal{X}$, is defined as follows:

$$\partial^\ell_\mathcal{X} = \frac{|\mathcal{V}_\mathcal{X}|}{\ell_\mathcal{X}} * 1000 \tag{3}$$

where $\ell_\mathcal{X}$ denotes the total number of LOC in clones of category $\mathcal{X}$.

Density of vulnerabilities per KLOC in type $\mathcal{T}$ code, denoted as $\partial^\ell_\mathcal{T}$, is defined as follows:

$$\partial^\ell_\mathcal{T} = \frac{|\mathcal{V}_\mathcal{T}|}{\ell_\mathcal{T}} * 1000 \tag{4}$$

where $\ell_\mathcal{T}$ denotes the total number of LOC in type $\mathcal{T}$ code.

Risk severity score per KLOC in category $\mathcal{X}$ clones, denoted as $\mathcal{R}_\mathcal{X}$, is defined as the ratio of sum of severity scores of the vulnerabilities found in clones of category $\mathcal{X}$ to the number of KLOC in the clones of category $\mathcal{X}$. Mathematically,

$$\mathcal{R}_\mathcal{X} = \frac{\sum_{\nu \in \mathcal{V}_\mathcal{X}} s(\nu)}{\ell_\mathcal{X}} * 1000 \tag{5}$$

where $s(\nu)$ denotes the severity score of vulnerability ν.

Risk severity score per KLOC in type $\mathcal{T}$ code, denoted as $\mathcal{R}_\mathcal{T}$, is defined as the ratio of sum of severity scores of the vulnerabilities found in type $\mathcal{T}$ code to the number of KLOC in that type of code. Mathematically,

$$\mathcal{R}_\mathcal{T} = \frac{\sum_{\nu \in \mathcal{V}_\mathcal{T}} s(\nu)}{\ell_\mathcal{T}} * 1000 \tag{6}$$

Density of a particular group of vulnerabilities $\mathcal{G}$ per KLOC in type $\mathcal{T}$ code, denoted as $\partial^\mathcal{G}_\mathcal{T}$, is calculated by dividing the number of vulnerabilities found in CWE category $\mathcal{G}$ in type $\mathcal{T}$ code by the total number of KLOC in that type of code. Mathematically,

$$\partial^\mathcal{G}_\mathcal{T} = \frac{|\mathcal{G}_\mathcal{T}|}{\ell_\mathcal{T}} * 1000 \tag{7}$$

where $|\mathcal{G}_\mathcal{T}|$ denotes the number of vulnerabilities belong to a CWE category $\mathcal{G}$ found in type $\mathcal{T}$ code.

III. Study Setup

The procedural steps of our empirical study are summarized in Figure 1.

A. Subject Systems

Our study investigates the source code of 34 open-source software systems written in C. Although some of the systems contain files written in other languages such as C++, Perl and other scripting languages, we only consider those files, which have extension '.c' or '.h' to exclude source code written in languages other than C. Projects of various sizes are deliberately chosen from different application domains including networking, communication, security, and text editing. Most of these subject systems are well-reputed in their respective development ecosystems (e.g., GitHub and SourceForge) and used in earlier research studies [8], [15], [41], [54], [55].

The names and sizes of the subject systems in LOC in clones and non-clone code are presented in Table II. In computation of sizes, only the source code written in C are considered.

TABLE II
Subject systems and their LOC in cloned and non-cloned code

Subject System	# of LOC written in C only		
	Non-clone	Clone	Total
Asn1c	40,487	5,488	45,975
Atlas	369,944	116,586	486,530
Clamav	337,019	40,371	377,390
Claws	239,784	32,868	272,652
Conky	27,759	14,494	42,253
Courier	107,223	13,105	120,328
Emacs	314,521	17,092	331,613
Ettercap	36,268	5,182	41,450
Ffdshow	832,342	46,367	878,709
Freediag	15,225	1,380	16,605
Freedroid	60,828	3,957	64,785
Gedit	42,112	4,331	46,443
Glimmer	29,536	4,923	34,459
Gnuplot	86,851	6,712	93,563
Gretl	302,777	36,499	339,276
Grisbi	99,205	18,479	117,684
Ipsec-tools	60,728	10,222	70,950
Modsecurity	26,332	7,232	33,564
Nedit	85,001	9,506	94,507
Net-snmp	231,422	42,725	274,147
Ocf-linux	45,761	8,743	54,504
Opendkim	47,298	18,480	65,778
Opensc	119,646	11,823	131,469
Putty	78,918	11,322	90,240
Razorback	36,403	8,823	45,226
Sdcc	3,477,010	4,122	3,481,132
Tboot	21,942	6,115	28,057
Tcl8	326,217	37,693	363,910
Tcpreplay	43,191	4,740	47,931
Trousers	56,673	17,599	74,272
Vi	21,924	734	22,658
Vim	314,480	5,752	320,232
XSupplicant	78,441	24,028	102,469
Zabbix	107,475	18,156	125,631

TABLE III
NiCad Settings For Code Clone Detection

Chosen Parameters for `NiCad`	Target Clone Types		
	Type-1	Type-2	Type-3
Dissimilarity Threshold	0.0	0.0	0.3
Identifier Renaming	no rename	blind rename	no rename
Granularity	block	block	block
Minimum Clone Size	5 LOC	5 LOC	5 LOC

The average size of the subject systems is 256 thousand LOC where the largest project consists of 3.4 million LOC and the smallest one contains 16 thousand LOC.

B. Code Clone Detection

Using the `NiCad` [39] clone detector (version 3.5), we separately detect code clones in each of the subject systems at the granularity of syntactic code blocks. The parameters settings of `NiCad` used in our study are mentioned in Table III. With these settings, `NiCad` detects *Type-1*, *Type-2*, and *Type-3* clones. Further details on `NiCad`'s tuning parameters and their influences on clone detection can be found elsewhere [39]. Then, we compute the *pure Type-2* and *pure Type-3* clones in accordance with their definitions outlined in Section II.

C. Security Vulnerability Detection

For the detection of vulnerabilities in source code, we use two open-source static analysis tools `Flawfinder` (version 1.3) [3] and `Cppcheck` (version 1.76.1) [2]. Their ability to detect different sets of vulnerabilities make them appropriate for our analysis. For example, `Flawfinder` is capable of detecting vulnerabilities such as *Uncontrolled Format String*,

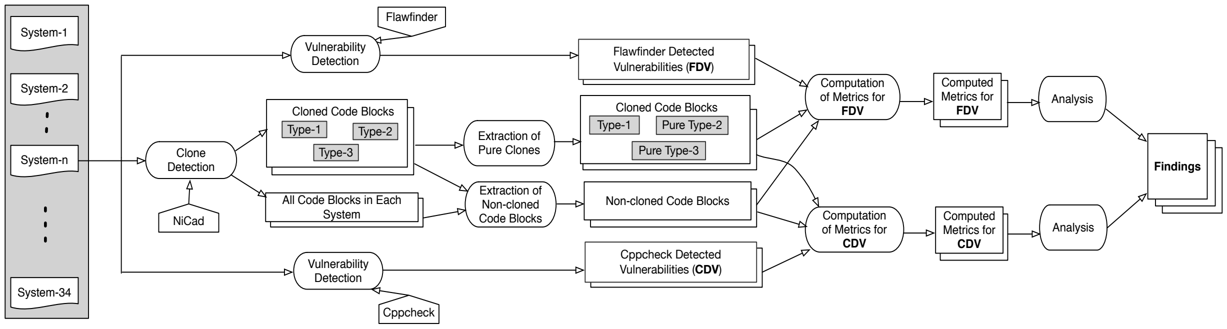

Fig. 1. Procedural Steps of the Empirical Study

Integer Overflow and *Use of Risky Cryptographic Algorithm*, which `Cppcheck` fails to detect [12]. On the other hand, `Cppcheck` is able to detect vulnerabilities such as *Memory Leak*, *Dead Code* and *Null Pointer Dereference* that `Flawfinder` cannot detect [12].

We now briefly describe how these tools work for the detection of security vulnerabilities and the ways these tools are used in our study. We also present justifications for choosing these tools for our study.

1) `Flawfinder`: The tool contains a database of common functions known to be vulnerable. It operates by performing lexical tokenization of the C/C++ code and comparing the tokens with those in the database. Once the comparison is performed, it reports a list of possible vulnerabilities along with a source code line number and a numeric risk-level (i.e., severity score) associated each of the detected vulnerabilities. The severity scores vary from one (indicating little security risk) to five (indicating high risk).

Although many other open-source static analysis tools such as, `RATS` [6], `SPLINT` [7] and `ITS4` [47] exist to identify potential security issues, we choose `Flawfinder` for a number of reasons. This tool is reported to have the highest vulnerabilities detection rate (i.e., highest recall) among all the existing security vulnerability detectors [12], [33], [48], [35]. Moreover, a comparison of vulnerability detection tools [33] also recommend choosing `Flawfinder` to detect security vulnerabilities. `Flawfinder` is also widely used in many earlier studies [35], [45], [48].

To detect vulnerabilities with `Flawfinder`, we execute the tool from command line interface and separately detect vulnerabilities in each of the subject systems in our study. We refer to the vulnerabilities detected using `Flawfinder` as $\mathcal{FDV}$ (`Flawfinder` Detected Vulnerabilities).

a) Limiting false positives in `Flawfinder`: Although `Flawfinder` has the highest detection rate, at times, it is blamed for reporting many false positives [48]. To reduce the false positives, we alter the default configuration of `Flawfinder`. We run the tool with *'-F'* configuration parameter that reduces 62% of the false positives as reported in a controlled experiment [44]. To further reduce the effect of false positives, we discard any detected vulnerabilities associated with risk-levels less than two.

TABLE IV
REDUCTION OF FALSE POSITIVES IN VULNERABILITY DETECTION USING THE CUSTOMIZED CONFIGURATION OF `FLAWFINDER`

Test Suite ID	**# of False Positives Reported**		**Reduction of False Positives**
	Default Config.	**Customized Config.**	
57	08	01	87.50%
58	10	04	60.00%

b) Effectiveness of customized configuration: To determine the effectiveness of the customized configuration stated above, we collect two C/C++ test suites, *test-suite-57* and *test-suite-58* from *Software Assurance Reference Dataset* (SARD) [4]. These test suites include 41 and 39 pieces of code respectively. While all pieces of code in *test-suite-57* are known to be vulnerable, none of the pieces of code in *test-suite-58* are vulnerable.

We run `Flawfinder` on the two test suites separately using both default and customized configurations. By comparing the reported vulnerabilities with the known vulnerabilities in the test suites, we compute false positives w.r.t. both test suites for each of the configurations and present the result in Table IV. Notice that with the customized configuration, false positives are reduced by 87.5% and 60% for *test-suite-57* and *test-suite-58* respectively. We also observe 30% reduction in the detection of vulnerabilities mostly due to the elimination of false positives using the customized configuration for *test-suite-57*. Thus, at the cost of a minor sacrifice in recall, the customized configuration is able to reduce significant number of false positives.

2) `Cppcheck`: `Cppcheck` supports a wide variety of static checks that are rigorous, rather than heuristic in nature [2]. `Cppcheck` is developed aims to report zero false positives [2], which makes it unique from other static security analysis tools. Unlike `Flawfinder`, `Cppcheck` does not assign numeric risk-levels to vulnerabilities. Instead, `Cppcheck` classifies vulnerabilities into six severity categories namely, *Error, Warning, Style, Performance, Portability,* and *Information*. We operate `Cppcheck` from command line using its default configuration separately on each of the subject systems. The output of the tool is generated in XML. We refer to the security vulnerabilities detected by `Cppcheck` as $\mathcal{CDV}$ (`Cppcheck` Detected Vulnerabilities).

Brief description along with CWE numbers of the major vulnerabilities detected in the subject systems using `Flawfinder` and `Cppcheck` are presented in Table I. Those

TABLE V
DETECTED SECURITY VULNERABILITIES AND THEIR SEVERITY IN CLONED AND NON-CLONED CODE

Subject System	**# of $\mathcal{FDV}$**		**# of $\mathcal{CDV}$**		**Severity score**	
	$\mathcal{NC}$	$\mathcal{C}$	$\mathcal{NC}$	$\mathcal{C}$	$\mathcal{NC}$	$\mathcal{C}$
Asn1c	119	10	183	18	309	70
Atlas	1,258	1,607	2,307	3,240	4,429	4,246
Clamav	1,151	166	6,379	1,901	2,689	345
Claws	429	35	1,496	138	1,148	108
Conky	310	69	10	41	850	200
Courier	2,569	270	1,152	67	7,479	900
Emacs	1,071	99	666	108	2,990	281
Ettercap	419	78	235	23	982	177
Ffdshow	1,306	158	10,042	424	1,457	330
Freediag	416	54	104	11	1,521	202
Freedroid	429	26	180	0	1,181	88
Gedit	21	9	276	7	51	18
Glimmer	151	19	326	214	450	64
Gnuplot	717	67	1,322	87	2,172	208
Gretl	2,850	359	2,992	249	8,847	1,116
Grisbi	54	40	293	27	177	82
Ipsec-tools	447	100	790	134	960	210
Modsecurity	163	22	193	40	308	44
Nedit	592	62	624	56	1,884	216
Net-snmp	1,715	352	2,002	177	4,071	763
Ocf-linux	153	17	379	61	245	134
Opendkim	201	40	296	98	383	146
Opensc	1,150	141	735	30	2,347	404
Putty	523	108	593	59	1,217	392
Razorback	104	31	359	39	261	94
Sdcc	26	9	291	38	182	32
Tboot	105	59	80	225	234	140
Tcl8	1,111	218	3,592	463	2,552	628
Tcpreplay	421	87	232	10	1,252	228
Trousers	263	110	460	34	560	224
Vi	144	4	403	16	417	52
Vim	805	22	2,336	43	2,264	70
XSupplicant	838	246	2,593	505	1,822	548
Zabbix	512	82	566	24	992	278

Here, $\mathcal{NC}$ = non-cloned code and $\mathcal{C}$ = cloned code

reported as vulnerabilities but do not have an associated CWE number are excluded from our analysis to further limit effects of possible false positives.

IV. ANALYSIS AND FINDINGS

After detecting clones and vulnerabilities in the software systems, we determine locations of the detected vulnerabilities in different types of code (i.e., cloned and non-cloned code). A vulnerability is said to be located in cloned code if the reported source code line number of that vulnerability included in a cloned block, otherwise, the vulnerability is located in non-cloned block. For each of the subject systems, we identify the co-locations of code clones and vulnerabilities, distinguish the vulnerabilities located in non-cloned portion of code, and compute all the metrics described in Section II. The number of $\mathcal{FDV}$, $\mathcal{CDV}$ in the clones and non-cloned code in each of the subject systems, and the cumulative vulnerability severity scores (obtained from `Flawfinder`) are presented in Table V.

We separately analyze the security vulnerabilities detected using both `Flawfinder` and `Cppcheck` to derive answers to the research questions RQ1, RQ2, and RQ5. Since `Cppcheck` does not provide the numeric severity score to indicate risk-level of a security vulnerability, the research questions RQ3 and RQ4 are addressed using $\mathcal{FDV}$ only.

Statisitical measurements. To verify the statistical significance of the results derived from our analyses, we apply the statistical *Mann-Whitney-Wilcoxon (MWW)* test [9] and *Kruskal-Wallis* test [9] at the significance level $\alpha = 0.05$. We

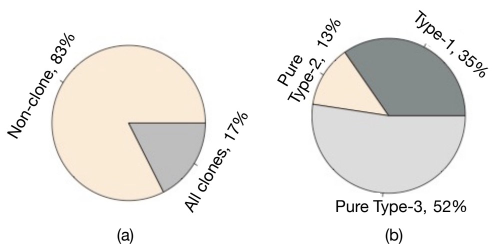

Fig. 2. Distribution of $\mathcal{FDV}$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

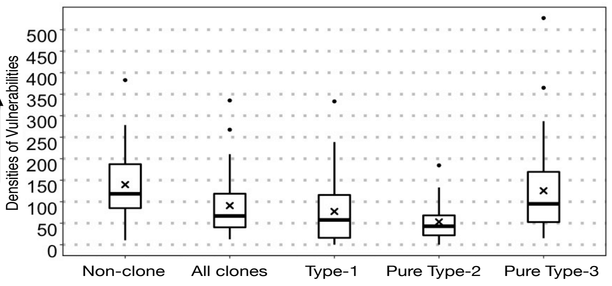

Fig. 3. Densities of $\mathcal{FDV}$ w.r.t. BOC

perform *Kruskalmc* [9] test for post-hoc analysis at the same significance level. As the non-parametric *MWW*, *Kruskal-Wallis* and *Kruskalmc* tests do not require normal distribution of data, those tests suit well for our purpose. To measure the effect size, we compute the non-parametric effect size *Cliff's delta* [9].

A. Vulnerabilities in Clones vs. Non-Cloned Code

Analysis Using $\mathcal{FDV}$: Figures 2(a) and 2(b) present the distributions of vulnerabilities detected using `Flawfinder` in non-cloned code and in different types of clones respectively over all the systems. As seen in Figure 2(a), 83% of all the vulnerabilities are found in non-cloned source code, whereas the clones contain only 17% of vulnerabilities.

The box-plot in Figure 3 presents the densities of vulnerabilities (i.e. $\mathcal{FDV}$) w.r.t. BOC (computed using Equation 1 and Equation 2) found in non-cloned code and in different types of clones over all the subject systems. The 'x' marks in the boxes indicate the mean densities over all the systems. As seen in Figure 3, the density of vulnerabilities (w.r.t. BOC) in non-cloned blocks is much higher than that in code clones.

It is highly probable that a larger portion of source code contains more vulnerabilities than a smaller portion of source code, which might be a reason why non-cloned code seems

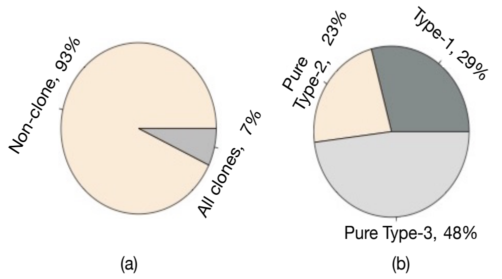

Fig. 4. Distribution of LOC (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

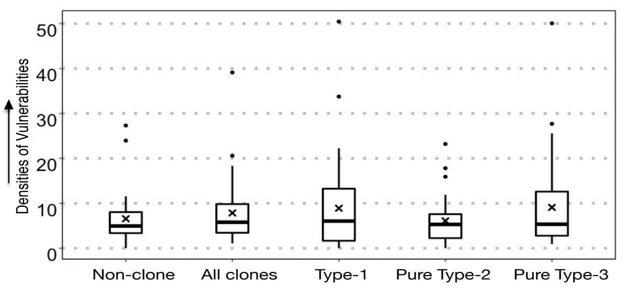

Fig. 5. Densities of $\mathcal{FDV}$ w.r.t. LOC

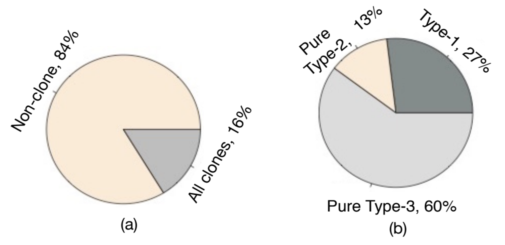

Fig. 6. Distribution of $\mathcal{CDV}$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

to have more vulnerabilities as observed in Figure 2(a) and Figure 3. To verify this possibility, we compute the distribution of LOC in non-cloned code and different types of clones over all the systems as presented in Figures 4(a) and 4(b) respectively. In Figure 4(a), we find that the number of LOC in non-cloned code is significantly higher compared to cloned code. We also compute the lengths of non-cloned and cloned code blocks in terms of average LOC over all the systems that are found to be 48.69 and 12.97 respectively. Thus, the possibility of influence of code size (in terms of LOC) on the number and density of vulnerabilities w.r.t. *BOC* is found to be true.

We, therefore, continue our investigations at a deeper level using the densities of vulnerabilities w.r.t. *LOC*. The box-plot in Figure 5 presents the densities of vulnerabilities w.r.t. LOC (computed using Equation 3 and Equation 4) found in non-cloned code and in different types of clones over all the subject systems. Figure 5 shows that both the median and average of densities of vulnerabilities (w.r.t. LOC) in cloned code (all clones) are higher compared to non-cloned code. We conduct a *MWW* test to measure the statistical significance of differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code. The p-value ($p = 0.20, p > \alpha$) obtained from the *MWW* test implies that observed differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code across all the subject systems are not statistically significant.

Analysis Using $\mathcal{CDV}$: Figures 6(a) and 6(b) present the distributions of $\mathcal{CDV}$ in non-cloned code and in different types of clones respectively over all the systems. Interestingly, the pattern of the distributions of vulnerabilities is similar to the pattern observed in Figures 2(a) and 2(b) drawn for $\mathcal{FDV}$.

Similar to Figure 5, Figure 7 presents the densities of vulnerabilities (i.e., $\mathcal{CDV}$) w.r.t. LOC (computed using Equation 3 and Equation 4) found in non-cloned code and in different types of clones over all the subject systems. As seen in Figure 7, the average of densities of vulnerabilities is higher in cloned code (all clones) compared to non-cloned code, although the median of densities of vulnerabilities is slightly higher in non-cloned code as opposed to code clones. Again, we conduct a *MWW* test to measure the statistical significance of differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code. The p-value ($p = 0.42, p > \alpha$) obtained from the statistical test implies that differences in the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code across all the subject systems do not differ significantly. This result agrees with that obtained for $\mathcal{FDV}$. Therefore, we derive the answer to the *RQ1* as follows:

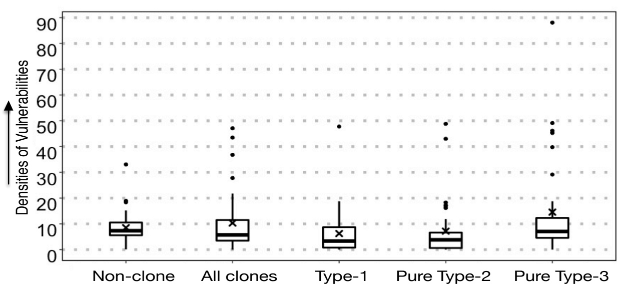

Fig. 7. Densities of $\mathcal{CDV}$ w.r.t. LOC

Ans. to RQ1: *Densities of vulnerabilities in cloned code are NOT significantly higher than non-cloned code.*

B. Densities of Vulnerabilities in Different Types of Clones

Analysis Using $\mathcal{FDV}$: The distribution of $\mathcal{FDV}$ portrayed in Figure 2(b) shows that the *pure Type-2* clones are found to have the minimum vulnerabilities whereas the number of vulnerabilities found in *Type-1* clones is higher than that in pure *Type-2* clones. The vulnerabilities found in cloned portion of source code are found to be dominated by those found in *pure Type-3* clones. However, the majority of cloned LOC are also in *pure Type-3* clones as can be observed in Figure 4(b), which might be a reason why a higher number of vulnerabilities are found in code clones of this particular category.

As seen in Figure 5, the densities of vulnerabilities w.r.t. *LOC* in different categories of code clones follow the same pattern of densities of vulnerabilities w.r.t. *BOC* where *pure Type-3* clones show the highest average density of vulnerabilities followed by *Type-1* clones, and *pure Type-2* clones show the lowest average density. Although noticeable differences are observed in the *averages* of densities of vulnerabilities, we do not see much differences in the *medians*. To determine the statistical significance of our observations, we conduct a *Kruskal-Wallis* test between the densities of $\mathcal{FDV}$ (w.r.t. LOC) of the three categories of clones. The p-value ($p = 0.5901, p > \alpha$) obtained from the *Kruskal-Wallis* test suggests no significant differences in the distributions of densities of vulnerabilities.

Analysis Using $\mathcal{CDV}$: As observed in Figure 2(b) and Figure 6(b), the patterns of distributions of both $\mathcal{FDV}$ and

$\mathcal{CDV}$ in different types of clones are very similar. However, a comparison of Figure 5 and Figure 7 makes some differences visible. In contrast with $\mathcal{FDV}$ (Figure 5), the average and median densities of $\mathcal{CDV}$ (Figure 7) in *Type-1* and *pure Type-2* code clones are almost equal while both the average and median are noticeably higher in *pure Type-3* clones. We also see that the *average* densities of both $\mathcal{FDV}$ and $\mathcal{CDV}$ are the highest in *pure Type-3* clones.

Again, to determine the significance of differences of densities of $\mathcal{CDV}$ in different types of clones, we conduct a *Kruskal-Wallis* test between the densities of $\mathcal{CDV}$ (w.r.t. LOC) of the three categories of clones. The p-value ($p = 0.008086, p < \alpha$) obtained from the *Kruskal-Wallis* test suggests significant differences in the distributions of densities of vulnerabilities. To determine the significance of the pairwise difference, we conduct a *Kruskalmc* post-hoc analysis. The test suggests statistical significance differences in the distributions of $\mathcal{CDV}$ in *pure Type-3* clones against *Type-1* and *pure Type-2* clones. The computed *Cliff's delta* d values 0.373 and 0.404 between *pure Type-3* and *Type-1* and between *pure Type-3* and *pure Type-2* respectively, indicate medium effect sizes. Based on our analyses of both $\mathcal{FDV}$ and $\mathcal{CDV}$, we now answer the *RQ2* as follows:

Ans. to RQ2: *Pure Type-3 clones are the most insecure category of clones, while Type-1 and pure Type-2 clones are almost equal in terms of security vulnerability.*

C. Severity of Security Risks in Cloned and Non-Cloned Code

Figures 8(a) and 8(b) present the distributions of cumulative severity scores of $\mathcal{FDV}$ in non-cloned code and in different types of clones respectively over all the systems. As seen in Figure 8(a), as much as 81% of total severity scores of $\mathcal{FDV}$ is associated with non-cloned code, which can be expected as non-cloned code contributes 93% of the entire source code (Figure 4(a)).

The box-plot in Figure 9 presents the risk severity scores per LOC (computed using Equation 5 and Equation 6) for non-cloned code and different types of clones for each of the subject systems. Figure 9 shows that both the median and average of the risk severity scores over all the systems in cloned code (all clones) are higher compared to non-cloned code. We conduct a *MWW* test to measure the statistical significance of these observed differences. The p-value ($p = 0.0494, p < \alpha$) obtained from the test indicates statistical significance of the differences. The computed *Cliff's delta* d value 0.381 suggests the effect size is medium. Therefore, we derive the answer to the RQ3 as follows:

Ans. to RQ3: *The security vulnerabilities in cloned code are significantly riskier than those in non-cloned code.*

D. Severity of Security Risks in Different Types of Clones

The distribution of cumulative severity scores of $\mathcal{FDV}$ in different types of code clones depicted in Figure 8(b) shows that vulnerabilities in *pure Type-2* clones have posed the lowest cumulative severity score whereas *pure Type-3* clones show the highest severity score and *Type-1* clones fit in between.

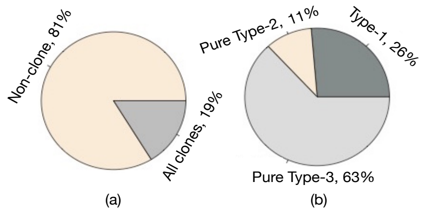

Fig. 8. Cumulative Severity Scores of $\mathcal{FDV}$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones

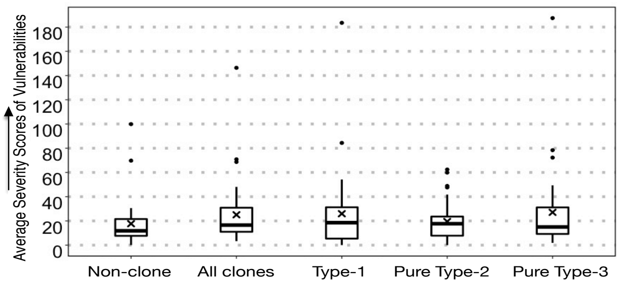

Fig. 9. Risk Severity Scores per KLOC in Cloned and Non-cloned Code

Again, Figure 9 shows that the *average* severity scores of vulnerabilities in *pure Type-3* and *Type-1* clones are almost equal while *pure Type-2* clones have slightly lower severity score compared to the former two. Moreover, we do not see much differences in the *medians*.

To determine the statistical significance of our observations, we conduct a *Kruskal-Wallis* test between *average* severity scores of vulnerabilities (w.r.t. LOC) of the three categories of clones. The p-value ($p = 0.5901, p > \alpha$) obtained from the *Kruskal-Wallis* test suggests no significant differences in the distributions of severities of vulnerabilities. Based on the findings, we now answer the *RQ4* as follows:

Ans. to RQ4: *There is no significant difference in the severity of security vulnerabilities found in different types of code clones.*

E. Frequently Encountered Categories of Vulnerabilities

Analysis Using $\mathcal{FDV}$: For each CWE category of $\mathcal{FDV}$ identified in the subject systems, we compute the densities of those separately for cloned code and non-cloned code in each of the subject systems (using Equation 7). Then we distinguish five CWE categories, which have the highest vulnerability densities in cloned code over all the subject systems. Let $\mathcal{D}_c$ denote the set of these five CWE categories. Similarly, we form another set $\mathcal{D}_{\bar{c}}$ consisting of five CWE categories of vulnerabilities having the highest densities in non-cloned code. By the union of these two sets, we obtain a set $\mathcal{D}$ of top six CWE categories of vulnerabilities that have the highest densities across both cloned and non-cloned code. Mathematically, $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_{\bar{c}}$.

Figure 10 presents the distributions of densities of these top six CWE categories of vulnerabilities in cloned and non-cloned code in each of the subject systems. As we see in Figure

TABLE VI
MWW TESTS OVER DENSITIES OF TOP SIX CWE CATEGORIES OF $\mathcal{FDV}$ PER KLOC IN CLONED AND NON-CLONED CODE

Categories of $\mathcal{FDV}$	P-Value	Significant?	*Cliff's delta* d
CWE-78	0.0233	Yes ($p < \alpha$)	0.384 (medium)
CWE-120	0.3264	No ($p > \alpha$)	not applicable
CWE-134	0.2809	No ($p > \alpha$)	not applicable
CWE-190	0.4681	No ($p > \alpha$)	not applicable
CWE-362	0.4051	No ($p > \alpha$)	not applicable
CWE-807	0.0012	Yes ($p < \alpha$)	0.379 (medium)

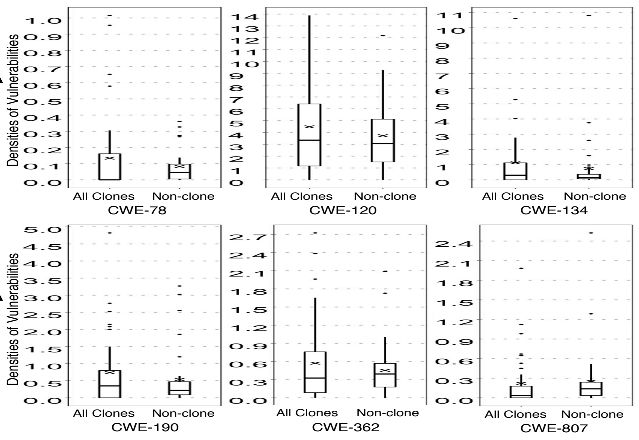

Fig. 10. Densities of top six CWE categories of $\mathcal{FDV}$ per KLOC

10, the *average* densities of the CWE-807 category of vulnerabilities is higher in non-cloned code compared to clones. The opposite is observed for the rest five CWE categories. For each of these six CWE categories, we separately conduct the *MWW* tests to determine the statistical significance of the differences in densities of vulnerabilities in cloned and non-cloned code. The resulted p-values of the tests presented in Table VI indicate that for CWE-78 and CWE-807 the differences are statistically significant and otherwise for the rest four CWE categories. Then, we compute the *Cliff's delta* d values for the CWE-78 and CWE-807 categories and find medium effect sizes for both. Thus, we can distinguish the CWE-78 category vulnerabilities as a category of vulnerabilities, which appear in cloned code more frequently than in non-cloned clone. We can also distinguish the CWE-807 category vulnerabilities having the opposite characteristics.

Analysis Using $\mathcal{CDV}$: Using similar procedure followed for $\mathcal{FDV}$, we obtain the set $\mathcal{D}$ of five CWE categories of vulnerabilities from $\mathcal{CDV}$. Figure 11 presents the distribution of densities of those five CWE categories of vulnerabilities over all the subject systems in cloned and none-cloned code. Again, for each of five CWE categories of vulnerabilities, we separately conduct *MWW* tests to determine the significance of differences in densities of $\mathcal{CDV}$ in clones and non-cloned code and also compute *Cliff's delta* d values for those distributions where $p < \alpha$.

The resulted p-values and computed effect sizes of those tests are presented in Table VII. Combining our observation in Figure 11, the p-values and the effect sizes in Table VII, we can infer that the densities of vulnerabilities of CWE-561 and CWE-686 categories are significantly higher with medium to large effect sizes in non-cloned code compared to cloned code.

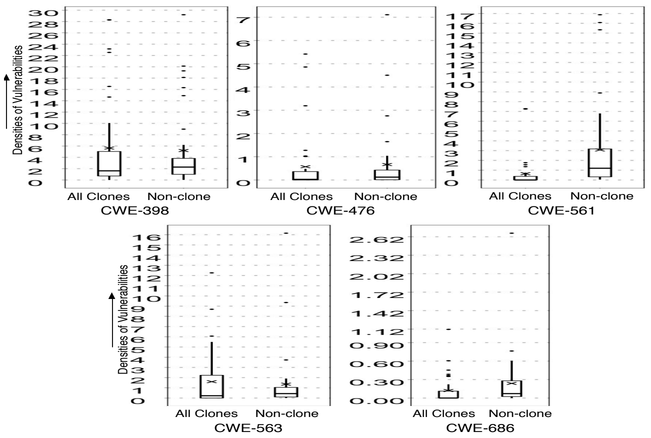

Fig. 11. Densities of top five CWE categories of $\mathcal{CDV}$ per KLOC

TABLE VII
MWW TESTS OVER DENSITIES OF TOP FIVE CWE CATEGORIES OF $\mathcal{CDV}$ PER KLOC IN CLONED AND NON-CLONED CODE

Categories of $\mathcal{CDV}$	P-Value	Significant?	*Cliff's delta* d
CWE-398	0.2980	No ($p > \alpha$)	not applicable
CWE-476	0.0630	No ($p > \alpha$)	not applicable
CWE-561	0.0321	Yes ($p < \alpha$)	0.692 (large)
CWE-563	0.1538	No ($p > \alpha$)	not applicable
CWE-686	0.0012	Yes ($p < \alpha$)	0.372 (medium)

Thus, we have been able to distinguish two CWE categories of vulnerabilities whose appearances are dominant in non-cloned code. Based on our analysis of both $\mathcal{FDV}$ and $\mathcal{CDV}$, we derive the answer to the *RQ5* as follows:

Ans. to RQ5: *It is possible to distinguish particular CWE categories of vulnerabilities, which frequently appear in cloned code (or non-cloned code).*

V. THREATS TO VALIDITY

Construct Validity: Although we have used two well-known tools for security vulnerability detection, we discarded from our study those reported vulnerabilities that the tools failed to associate with a CWE enumeration. We deliberately excluded those because a so-called vulnerability without an associated CWE number can actually be a stylistic issue that the tools erroneously report as a security vulnerability. In the selection of the two dominant sets of CWE categories of vulnerabilities (Section IV-E), we picked top five CWE categories of vulnerabilities for each set having the highest densities in cloned and non-cloned code. Although those chosen vulnerabilities cover more than 85% instances of vulnerabilities found in all the systems, this choice may still be considered as a threat to validity of this work. The computation of averages of the ordinal values of severity scores in the analyses for *RQ3* and *RQ4* can be questioned, although it serves our purpose of analyzing comparative severities of groups of vulnerabilities.
Internal Validity: While detecting vulnerabilities, false positives and false negatives could be two major threats to validity of this study. Hence, for vulnerability detection, we have used two different tools, `Flawfinder` [3] and `Cppcheck` [2]. `Flawfinder` is known to have high recall [12], [33], [35], [48], and `Cppcheck` is reputed for its high precision with zero

false positives [2]. Moreover, while using `Flawfinder`, we used a customized configuration, which significantly minimize the false positives [44], as also demonstrated in Section III-C of this paper. In addition, some types of vulnerabilities (e.g., *Null Pointer Dereference*) may span over multiple statements, which may partly overlap with both cloned and non-cloned code. Such a phenomenon couldn't be captured in the study as the vulnerability detection tools report only a line number in source file indicating the location of a particular vulnerability detected.

The clone detector, `NiCad` [39], used in our study, is reported to be very accurate in clone detection [39], and we have carefully set `NiCad`'s parameters for the detection of *Type-1*, *Type-2*, and *Type-3* clones. Moreover, we have taken care to avoid double-counting of nested blocks and vulnerabilities identified in them. In addition, we have manually verified the correctness of computations for all the metrics used in our work. Thus, we develop high confidence in the internal validity of this study.

External Validity: Although our study includes a large number of subject systems, all the systems are open-source and written in C. Thus the findings from this work may not be generalizable for industrial systems and source code written in languages other than C.

Reliability: The methodology of this study including the procedure for data collection and analysis is documented in this paper. The subject systems being open-source, are freely accessible while the tools `Flawfinder`, `Cppcheck` and `NiCad` are also available online. Therefore, it should be possible to replicate the study.

VI. Related Work

As mentioned before, no other work in the literature include a comparative study of security vulnerabilities in code clones and non-cloned source code. Hence, the studies which examined the characteristics and impacts of code clones are considered relevant to our work. Some studies included a comparative analysis of certain characteristics in clones against non-cloned code, as discussed below.

Lozano and Wermelinger [31] analyzed revisions of only four open-source projects and suggested that having a clone may increase the maintenance effort for changing a method. Hotta et al. [18] studied the changeability of cloned and non-cloned code in revisions of 15 open-source software systems. They reported code clones not to have any negative impact on software changeability, which contradicts the claim of Lozano and Wermelinger [31].

Towards understanding the stability of code clones, Lozano et al. [32] studied changes clones across revisions of only one software system and reported that a vast majority of methods experience larger and frequent changes when they contain cloned code. Based on a study on the revisions of 12 software systems, Mondal et al. [34] also reported code clones to be less stable. However, opposite results are reported from the other studies [10], [16], [17], [27]. In another study, Sajnani et al. [42] attempted to identify the relationships of code clones with statically identified bugs in systems written in Java. They found considerably lower number of bugs in code clones compared to non-cloned code.

Recently, Islam and Zibran [22] compared a large comparative study of the code 'vulnerabilities' in cloned and non-cloned code. In their work, 'vulnerability' was defined as the "problems in the source code identified based on bad coding patterns i.e, code smells". Our study is inspired from their work and significantly differs from theirs. First, in contrast with their study of code smells, we have studied the real security flaws widely known as security vulnerabilities. Second, they studied software systems written in Java, while we have studied systems written in C. Third, they used *PMD* [5] to detect code smells, whereas we have used two separate tools *Flawfinder* and *Cppcheck* to detect security vulnerabilities in source code. In addition, we have analyzed the severities of security vulnerabilities, while such severity of code smells was not studied in the aforementioned work of Islam and Zibran.

Attempts are also made to explore fault-proneness of clones by relating them with bug-fixing changes obtained from commit history. Such a study was conducted by Jingyue et al. [30], who reported that only 4% of the bugs were found in duplicated code. In a similar study, Rahman et al. [36] also observed that majority of bugs were not significantly associated with clones. Another study [21] along the same line reported that 55% of bugs in cloned code can be replicated bugs.

As discussed before, the contradictory results are often reported from comparative studies between clones and non-cloned code. This implies the necessity of further comparative investigations from a different dimension, which is exactly what we have done in this study. We have carried out a comparative investigation of *security vulnerabilities* in clones and non-coned code, which was missing in the literature. In addition, the comparative analysis of vulnerabilities in *Type-1*, *pure Type-2*, and *pure Type-3* clones is another important aspect of our work.

VII. Conclusion

In this paper, we have presented a large quantitative empirical study of the security vulnerabilities in clones and non-cloned code clones in 34 open-source software systems (8.7 million LOC) written in C. To the best of our knowledge, no such studies exists in the literature that performed a comparative analysis of security vulnerabilities in cloned and non-cloned code. For the detection of security vulnerabilities in source code, we have used two different tools (`Flawfinder` and `Cppcheck`), one of which is known to have high recall while the other is reputed for its high precision. For clone detection, we used a state-of-the-art clone detector, `NiCad`, which is also reported to have high accuracy.

Our study reveals that the security vulnerabilities found in code clones have higher severity of security risks compared to those in non-cloned code. However, the proportion (i.e., density) of vulnerabilities in clones and non-cloned code does not have any significant difference. Among the three categories (i.e., *Type-1*, *pure Type-2*, and *pure Type-3*) of code clones studied in our work, *pure Type-3* clones are found to be the most insecure whereas *Type-1* and *pure Type-2* clones are nearly equal in terms of the vulnerabilities found in them. The results are validated in the light of statistical significance.

The findings from this study advance our understanding of the characteristics, impacts, and implications of code clones in software systems. These findings will help in identifying problematic clones, which demand extra care and those vulnerabilities about which the developers need to be particularly cautious about while reusing code by cloning.

In future, we plan to conduct qualitative analyses to advance our understanding of such results. Moreover, we plan to perform similar analyses using software systems written in languages other than C to verify to what extent the findings from this study also applies to a broader range of source code written in diverse programming languages.

REFERENCES

[1] *Common Weakness Enumeration*. https://cwe.mitre.org, July 2017.

[2] *Cppcheck - A tool for static C/C++ code analysis*. http://cppcheck.sourceforge.net, July 2017.

[3] *Flawfinder - C/C++ Source Code Analyzer*. http://www.dwheeler.com/flawfinder/, July 2017.

[4] *NIST Software Assurance Reference Dataset*. https://samate.nist.gov/SRD/, July 2017.

[5] *PMD - Source Code Analyzer*. http://pmd.sourceforge.net, July 2017.

[6] *RATS - Rough Auditing Tool for Security*. http://www.securesw.com/rats/, July 2017.

[7] *SPLINT - Secure Programming LINT*. http://www.splint.org, July 2017.

[8] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. pages 376–385, 2005.

[9] D. Anderson, D. Sweeney, and T. Williams. *Statistics for Business and Economics*. Thomson Higher Education, 10th edition, 2009.

[10] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *CSMR*, pages 81–90, 2007.

[11] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *ICSM*, pages 273–282, 2011.

[12] H. Brar and P. Kaur. Comparing detection ratio of three static analysis tools. *Int. Journal of Computer Applications*, 124(13):35–40, 2015.

[13] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109 –118, 1999.

[14] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[15] M. Gabel and Z. Su. A study of the uniqueness of source code. In *FSE*, pages 147–156, 2010.

[16] N. Göde and J. Harder. Clone stability. In *CSMR*, pages 65–74, 2011.

[17] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *ICSE*, pages 311–320, 2011.

[18] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in softw. evolution?: an emp. study on open source softw. In *IWPSE-EVOL*, pages 73–82, 2010.

[19] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of finding inconsistently-changed bugs in code clones of mobile software. In *IWSC*, pages 94–95, 2012.

[20] Research Triangle Institute. The economic impacts of inadequate infrastructure of software testing. RTI Project Report 7007.011, National Institute of Standards and Technology, 2002.

[21] J. Islam, M. Mondal, and C. Roy. Bug replication in code clones: An empirical study. In *SANER*, 2016.

[22] M. Islam and M. Zibran. A comparative study on vulnerabilities in categories of clones and non-cloned code. In *IWSC*, pages 8–14, 2016.

[23] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *SSP*, pages 48–62, 2012.

[24] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE*, pages 55–64, 2007.

[25] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495, 2009.

[26] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *TESO*, pages 443–446, 2008.

[27] J. Krinke. Is cloned code more stable than non-cloned code? In *SCAM*, pages 57–66, 2008.

[28] J. Krinke. Is cloned code older than non-cloned code? In *IWSC*, pages 28–33, 2011.

[29] 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.

[30] J. Li and M. Ernst. CBCD: Cloned buggy code detector. In *ICSE*, pages 310–320, 2012.

[31] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM*, pages 227–236, 2008.

[32] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based exp. In *MSR*, pages 18–21, 2007.

[33] R. McLean. Comparing static security analysis tools using open source software. In *SSRC*, pages 68–74, 2012.

[34] M. Mondal, C. Roy, and K. Schneider. An empirical study on clone stability. *ACM Applied Computing Review*, 12(3):20–36, 2012.

[35] D. Pozza, R. Sisto, L. Durante, and A. Valenzano. Comparing lexical analysis tools for buffer overflow detection in network software. In *Comsware*, pages 126–133, 2006.

[36] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? In *MSR*, pages 72–81, 2010.

[37] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.

[38] C. Roy and J. Cordy. A survey on software clone detection research. Tech Report TR 2007-541, Queens University, Canada, 2007.

[39] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[40] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future. In *CSMR-18/WCRE-21 Software Evolution Week (SEW'14)*, pages 18–33, 2014.

[41] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.

[42] H. Sajnani, V. Saini, and C. A comparative study of bug patterns in java cloned and non-cloned code. In *SCAM*, pages 21–30, 2014.

[43] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *WCRE*, pages 13–21, 2010.

[44] A. Sotirov. Automatic vulnerability detection using static source code analysis. Master's thesis, The University of Alabama, 2005.

[45] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *SS*, pages 365–377, 2008.

[46] R. Tonder and C. Goues. Defending against the attack of the micro-clones. In *ICPC*, pages 1–4, 2016.

[47] J. Viega, J. Bloch, T. Kohno, and G. Macgraw. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security*, 5(3):238–261, 2002.

[48] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, pages 124–138, 2003.

[49] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *MSR*, pages 149–158, 2013.

[50] M. Zibran and C. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105–114, 2011.

[51] M. Zibran and C. Roy. The road to software clone management. Technical Report 2012-03, University of Saskatchewan, Canada, 2012.

[52] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring. *IET Software*, 7(3):167–186, 2013.

[53] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *ICECCS*, pages 295–304, 2011.

[54] M. Zibran, R. Saha, C. Roy, and K. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *ACM-SAC (SE Track)*, pages 1123–1130, 2013.

[55] M. Zibran, R. Saha, C. Roy, and K. Schneider. Genealogical insights into the facts and fictions of clone removal. *ACM Applied Computing Review*, 13(4):30–42, 2013.