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 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 causes the multiplication and propagation of program faults and security vulnerabilities [22], [25], [51]. 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], [51]. Nevertheless, software systems typically have 9%-17% [52] 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 bug-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...
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>
<thead>
<tr>
<th>Security Vulnerability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Overflow</td>
<td>An application attempts to write data past the end of a buffer (CWE-120).</td>
</tr>
<tr>
<td>Use/Null Pointer</td>
<td>Dereferencing a pointer that is null (CWE-476).</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>The result of an arithmetic operation exceeds the maximum size of the integer type (CWE-561).</td>
</tr>
<tr>
<td>Memory Leak</td>
<td>Not releasing allocated memory (CWE-401).</td>
</tr>
<tr>
<td>Double Free</td>
<td>Access memory after it has been freed (CWE-416).</td>
</tr>
<tr>
<td>ritch</td>
<td>An application attempts to write data past the end of a buffer (CWE-120).</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>An application attempts to write data past the end of a buffer (CWE-120).</td>
</tr>
<tr>
<td>Use/Null Pointer</td>
<td>Dereferencing a pointer that is null (CWE-476).</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>The result of an arithmetic operation exceeds the maximum size of the integer type (CWE-561).</td>
</tr>
<tr>
<td>Memory Leak</td>
<td>Not releasing allocated memory (CWE-401).</td>
</tr>
<tr>
<td>Double Free</td>
<td>Access memory after it has been freed (CWE-416).</td>
</tr>
<tr>
<td>ritch</td>
<td>An application attempts to write data past the end of a buffer (CWE-120).</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>An application attempts to write data past the end of a buffer (CWE-120).</td>
</tr>
<tr>
<td>Use/Null Pointer</td>
<td>Dereferencing a pointer that is null (CWE-476).</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>The result of an arithmetic operation exceeds the maximum size of the integer type (CWE-561).</td>
</tr>
<tr>
<td>Memory Leak</td>
<td>Not releasing allocated memory (CWE-401).</td>
</tr>
<tr>
<td>Double Free</td>
<td>Access memory after it has been freed (CWE-416).</td>
</tr>
</tbody>
</table>

Type-1 Clones: Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called Type-1 clones [50].

Type-2 Clones: Type-2 clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout, and comments [50].

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 [50].

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

### Density of vulnerabilities w.r.t. BOC in category $\lambda$ clones

Let, $\delta_X$ is defined as the ratio of the number of vulnerabilities found in clones of category $\lambda$ and the number of cloned blocks in category $\lambda$ clones. Mathematically,

$$\delta_X = \frac{|\mathcal{V}_\lambda|}{\beta_X}$$

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

### Density of vulnerabilities w.r.t. BOC in type $T$ code

Let, $\delta_T$, is defined as the ratio of the number of vulnerabilities found in type $T$ code and total number of blocks in type $T$ code, where $T \in \{C, \bar{C}\}$. Mathematically,

$$\delta_T = \frac{|\mathcal{V}_T|}{\beta_T}$$

where $|\mathcal{V}_T| \in \{\mathcal{V}_C, \mathcal{V}_\bar{C}\}$ and $\beta_T$ denotes the total number of blocks in type $T$ code.
Density of vulnerabilities per 1,000 LOC (KLOC) in category $\mathcal{X}$ clones, denoted as $\partial_{\mathcal{X}}^\ell$, is mathematically defined as follows:

$$\partial_{\mathcal{X}}^\ell = \frac{|\mathcal{X}|}{\ell_{\mathcal{X}}} \times 1000$$

(3)

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

Density of vulnerabilities per KLOC in type $T$ code, denoted as $\partial_T^\ell$, is mathematically defined as follows:

$$\partial_T^\ell = \frac{|\mathcal{V}_T|}{\ell_T} \times 1000$$

(4)

where $\ell_T$ denotes the total number of LOC in type $T$ code.

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

$$\mathcal{R}_X = \frac{\sum_{\nu \in \mathcal{V}_X} s(\nu)}{\ell_{\mathcal{X}}} \times 1000$$

(5)

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

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

$$\mathcal{R}_T = \frac{\sum_{\nu \in \mathcal{V}_T} s(\nu)}{\ell_T} \times 1000$$

(6)

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

$$\partial_{\mathcal{T}}^\mathcal{G} = \frac{|\mathcal{G}_T|}{\ell_T} \times 1000$$

(7)

where $|\mathcal{G}_T|$ denotes the number of vulnerabilities belong to a CWE category $\mathcal{G}$ found in type $T$ 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. Various sizes’ projects 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], [53].

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. 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.

<table>
<thead>
<tr>
<th>Subject System</th>
<th># of LOC written in C only</th>
<th>Non-clone</th>
<th>Clone</th>
<th>Total</th>
<th>Non-clone</th>
<th>Clone</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asnlc</td>
<td>40,487</td>
<td>5,485</td>
<td>45,975</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atlas</td>
<td>369,944</td>
<td>116,586</td>
<td>486,530</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clamav</td>
<td>337,019</td>
<td>40,371</td>
<td>377,390</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Claws</td>
<td>239,784</td>
<td>32,868</td>
<td>272,652</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comfy</td>
<td>217,795</td>
<td>14,494</td>
<td>232,289</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Courier</td>
<td>107,223</td>
<td>13,105</td>
<td>120,328</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Emacs</td>
<td>314,521</td>
<td>17,092</td>
<td>331,613</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ettercap</td>
<td>36,268</td>
<td>7,182</td>
<td>43,450</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filthow</td>
<td>832,342</td>
<td>46,367</td>
<td>878,709</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Freedig</td>
<td>15,233</td>
<td>1,390</td>
<td>16,623</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Freedroid</td>
<td>60,828</td>
<td>3,957</td>
<td>64,785</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gedit</td>
<td>42,112</td>
<td>4,331</td>
<td>46,443</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Glimmer</td>
<td>29,536</td>
<td>4,923</td>
<td>34,459</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gimpool</td>
<td>80,831</td>
<td>8,712</td>
<td>89,543</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gremln</td>
<td>302,777</td>
<td>36,499</td>
<td>339,276</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goochi</td>
<td>99,205</td>
<td>18,479</td>
<td>117,684</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iпsectools</td>
<td>60,728</td>
<td>10,222</td>
<td>70,950</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modsecurity</td>
<td>26,332</td>
<td>7,232</td>
<td>33,564</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ncedit</td>
<td>51,001</td>
<td>5,806</td>
<td>56,807</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Net-smp</td>
<td>231,422</td>
<td>42,725</td>
<td>274,147</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orc-fh</td>
<td>45,761</td>
<td>8,743</td>
<td>54,504</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sccn</td>
<td>47,298</td>
<td>16,480</td>
<td>63,778</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opensec</td>
<td>119,646</td>
<td>11,823</td>
<td>131,469</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td>78,914</td>
<td>11,390</td>
<td>90,304</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Razorback</td>
<td>36,405</td>
<td>5,823</td>
<td>42,228</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sdec</td>
<td>3,477,010</td>
<td>4,122</td>
<td>3,481,132</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tbus</td>
<td>21,942</td>
<td>6,115</td>
<td>28,057</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tebit</td>
<td>326,217</td>
<td>37,693</td>
<td>363,910</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teepreplay</td>
<td>43,191</td>
<td>4,740</td>
<td>47,931</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Troubles</td>
<td>36,673</td>
<td>17,599</td>
<td>54,272</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vt</td>
<td>211,294</td>
<td>734</td>
<td>212,028</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vim</td>
<td>314,480</td>
<td>5,752</td>
<td>320,232</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XApplicant</td>
<td>79,441</td>
<td>24,028</td>
<td>103,469</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zabbix</td>
<td>107,979</td>
<td>18,156</td>
<td>126,135</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, 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 Mem-
ory 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 [48], [35], [45].

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 $FDV$ (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.

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 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 aiming 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 $CDV$ (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 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 code. For each of the subject systems, we identify the co-locations of code clones and vulnerabilities, distinguish the vulnerabilities located in cloned code and in different types of clones, and compute all the metrics described in Section II. The number of FDV, 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 FDV only.

Statistical 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 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 employ the non-parametric effect size Cliff’s delta [9].

A. Vulnerabilities in Clones vs. Non-Cloned Code

Analysis Using 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 detected using FDV in non-cloned code and in different types of clones respectively. The ‘x’ marks in the boxes indicate the mean densities over all the subject systems. As seen in Figure 3, the density of vulnerabilities (w.r.t. BOC) 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 code and in different types of clones respectively.

Indeed, a larger portion of source code is likely to contain more vulnerabilities than a smaller portion of source code, which might be a reason why non-cloned code seems to have more vulnerabilities as observed in Figure 2(a) and Figure 3.

![Fig. 2. Distribution of $FDV$ (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones](image1)

![Fig. 3. Densities of $FDV$ w.r.t. BOC](image2)

![Fig. 4. Distribution of LOC (a) in Cloned and Non-cloned Code and (b) in Different Types of Clones](image3)
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 is 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 ($P = 0.42, P > \alpha$) 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 $FDV$. Therefore, we derive the answer to the RQ1 as follows:

**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 $FDV$:** The distribution of $FDV$ 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 $FDV$ (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 $CDV$:** As observed in Figure 2(b) and Figure 6(b), the patterns of distributions of both $FDV$ and $CDV$ 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 Kruskal-Wallis 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. 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 \( D_c \) denote the set of these five CWE categories. Similarly, we form another set \( D_{cc} \) 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 \( D \) of top six CWE categories of vulnerabilities that have the highest densities across both cloned and non-cloned code. Mathematically, \( D = D_c \cup D_{cc} \).

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 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 CDV: Using similar procedure followed for FDV, we obtain the set D of five CWE categories of vulnerabilities from 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 CDV in clones and non-cloned code and also compute Cliff’s delta d values for those distributions where (P < α).

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. 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 FDV and 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 which 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 [33], [12],
clones to be less stable. However, opposite results are reported on 12 software systems, Mondal et al. [34] also reported code clones are often not present. Based on a study on the revisions of 15 open-source software systems, they concluded that clones are not a significant factor in software changeability, which contradicts the claim of Lozano and Wermelinger [31].

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 include 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], [17], [16], [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 Jinyue 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-clone 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 substantially 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