A Comparative Study on Vulnerabilities in Categories of Clones and Non-Cloned Code

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

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

Abstract—Code clones are serious code smells. To investigate bug-proneness of clones as opposed to clone-free source code, earlier attempts have studied the stability of code clones and their contributions to program faults. This paper presents a comparative study on different types of clones and non-cloned code on the basis of their vulnerabilities, which may lead to software defects and issues in future. The empirical study along this new dimension examines source code of 97 software systems and derives results based on quantitative analysis with statistical significance. The findings from this work add to our understanding of the characteristics and impacts of clones, which can be useful in clone-aware software development and in devising techniques for minimizing the negative impacts of code clones.

I. INTRODUCTION

Source code reuse by copy-paste is a common practice that software developers adopt to increase productivity. Such a reuse mechanism typically results in duplicate or very similar code fragments commonly known as code clones. Aside from such deliberate cloning, unintentional clones are also created for various reasons under diverse circumstances [25], [27]. Software systems typically have 9%-17% [33] cloned code, and the proportion is sometimes found to be even 50% [26] or higher [5].

Despite the few benefits [16] of cloning, code clones are detrimental in most cases [14], [16], [20]. Code clone is a notorious code smell (i.e., a symptom indicating source of future problems) [7], that cause serious problems such as reduced code quality, code inflation, program faults, security vulnerabilities, and bugs propagation [14], [32]. Clones are thus a major contributor to the high maintenance cost for software systems, and as much as 80% of software costs are spent on maintenance [12]. Therefore, it is necessary to keep the number of clones at the minimum and to remove them from source code by refactoring. However, not all the clones in a software system are harmful [16], neither it is feasible to remove all the clones in source code by refactoring [4], [32]. Therefore, we must distinguish the context and characteristics of clones, which make them malign as opposed to the benign clones.

Towards this goal, several studies have been performed in the past to examine or exploit comparative stability of clones as opposed to non-cloned code [9], [10], [17], [18], [22], relationships of clones with bug-fixing changes [3], [11], [13], [14], [15], [19], [24], [29], [31], and the impacts of clones on program’s changeability [8], [20], [21]. Note that, code smells are symptoms of poor coding patterns and are probable sources of future serious problems. While code clone itself is a notorious code smell [7] other fine grained code smells, defects and vulnerabilities often hide inside cloned code. Thus, such “smelly code clones” can be regarded as bug-prone clones, which are likely to cause serious defects, and the reuse by copy-pasting of such a bug-prone piece of code causes multiplication of bug-proneness elsewhere in the software system.

Earlier attempts [13], [15], [19], [31] to determine bug-proneness of code clones relied on long-term history of bug fixing changes preserved in version control system. While such studies make important contributions, their approaches do not fit well for proactive clone management, especially at the early stages of software development process where significantly long history of bug-fixing changes are not available. Therefore, to determine bug-proneness of code clones, we choose to apply static source code analysis techniques that do not require any bug-fixing history.

This paper presents an empirical study on the relationships of program vulnerabilities with code clones. Here ‘vulnerability’ refers to problems in the source code identified based on bad coding patterns [7], which lead to bugs, security holes, performance issues, design flaws, and other difficulties. A list of such vulnerabilities are presented in Table I. A particular piece of code is considered vulnerable if it contains code smells or bad coding patterns, and the severity of the vulnerability is dictated by the severity of existing code smells. In this work, we address the following three research questions.

RQ1: Are code clones more vulnerable than non-cloned code or vice versa? — Rahman et al. [24] reported that the great majority of software defects are not significantly associated with clones, while Juergens et al. [14] claimed otherwise.

RQ2: Are clones of a certain category relatively more vulnerable than others? — If a certain type of clones are found to be more vulnerable, those clones can be high-priority candidates for removal or careful maintenance.

RQ3: Is there a particular set of vulnerabilities that appear more frequently in cloned code as opposed to non-cloned code? — If such a set of vulnerabilities can be identified, the findings will help software developers staying cautious of such vulnerabilities while cloning source code. In addition, those particular set of vulnerabilities can be avoided or removed by the use of clone refactoring.
To answer the aforementioned research questions, we conduct a quantitative empirical study over 97 open-source software systems drawn from diverse application domains. Using a wide range of metrics and characterization criteria, we carry out an in-depth analysis on the source code of the systems with respect to different categories of code clones, non-clone code, and a diverse set of vulnerabilities.

II. TERMINOLOGY AND METRICS

In this section, we describe and define the terminologies and metrics used in our work.

A. Characterizing Terminologies

Our study includes clones at the granularity of syntactic blocks at different levels of similarities.

Type-1 Clones: Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called Type-1 clones [27].

Type-2 Clones: Type-2 clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout and comments [27].

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 [27].

Notice that 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, we further define two subsets of Type-2 and Type-3 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.

B. Metrics

The most important metrics used in this study 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). Note that only source lines of code are taken into consideration excluding comments and blank lines.

Density of vulnerabilities w.r.t. BOC in all clones, denoted as $\partial^\alpha_t$, is defined as the ratio of the number of vulnerabilities found in all clones and total the number of all the clones. Mathematically,

$$\partial^\alpha_t = \frac{\nu c}{\beta c}$$ (2)

where $\nu c$ denotes the number of vulnerabilities found in all clones and $\beta c$ denotes the total number of all categories of clones.

Density of vulnerabilities w.r.t. BOC in non-cloned code, denoted as $\partial^\alpha_n$, is defined as the ratio of the number of vulnerabilities found in non-cloned code and the number of non-cloned blocks of code. Mathematically,

$$\partial^\alpha_n = \frac{\nu n}{\beta n}$$ (3)

where $\nu n$ denotes the number of vulnerabilities found in non-cloned code and $\beta n$ denotes the total number of non-cloned blocks of code.

Density of vulnerabilities w.r.t. LOC in category $x$ clones, denoted as $\partial^\beta_x$, is defined as the ratio of the number of vulnerabilities found in clones of category $x$ and the number of LOC in all the clones of category $x$ where, category $x \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$. Mathematically,

$$\partial^\beta_x = \frac{\nu x}{\ell x}$$ (4)

where $\nu x$ denotes the number of vulnerabilities found in clones of category $x$ and $\ell x$ denotes the total number of LOC in clones of category $x$.

Density of vulnerabilities w.r.t. LOC in all clones, denoted as $\partial^\beta_q$, is defined as the ratio of the number of vulnerabilities found in all clones and the number of LOC in all the clones. Mathematically,

$$\partial^\beta_q = \frac{\nu q}{\ell q}$$ (5)

where $\nu q$ denotes the number of vulnerabilities found in all clones and $\ell q$ denotes the total number of LOC in all clones.

Density of vulnerabilities w.r.t. LOC in non-cloned code, denoted as $\partial^\beta_r$, is defined as the ratio of the number of vulnerabilities found in non-cloned code and the number of LOC in all non-cloned blocks of code. Mathematically,

$$\partial^\beta_r = \frac{\nu r}{\ell r}$$ (6)

where $\nu r$ denotes the number of vulnerabilities found in non-cloned code and $\ell r$ denotes the total number of LOC in non-cloned blocks of code.

Density of a particular vulnerability $v$ in cloned code, denoted as $d_c(v)$, is calculated by dividing the number of instances of $v$ found in cloned code by the total number of LOC in cloned code. Mathematically,

$$d_c(v) = \frac{\nu v}{\ell c}$$ (7)

where $\nu v$ denotes the number of instances of vulnerability $v$ found in cloned code and $\ell c$ denotes the total number of LOC across all clones.
D. A Variant of Cyclomatic complexity, which treats switch statements as a single decision point.

Non-final method argument that is never assigned to.

detects Local definitions and references to variables on different paths.

Use of higher level exception in catching low level error conditions.

Useless parentheses in code.

Type-2 Unnecessary explicit initialization of class’s member fields.

NiCad Class’s private fields whose values never change once they are initialized but not made final.

F error message.

A field, local, or parameter with a too short name.

[28] clone detector (version 3.5), we

Missing annotations for methods.

V S Class’s member variables not marked as transient, static, or missing accessor methods.

found in non-cloned code by the total number

Local variable assigned only once but not declared final.

Use of hard coded literals in conditional statements.

Program unit needing too much knowledge about other units.

TUDY Use of negation within an ‘if’ expression in ‘if-else’ statement.

‘if’ statements without accompanying curly braces.

Assignment of a “null” to a variable (outside of its declaration).

Description denotes the total number of

Method with more than one exit points.

NiCad Mains and written in Java.

collection of open source systems of diverse application do-

systems of the Qualitas Corpus [30], which is a large curated

A. Subject Systems

Our study investigates the source code of 97 software

systems of the Eclipse plugin variant of the tool. All others

parameters of NiCad are set to the defaults. Using NiCad, we

separately detect code clones (with at least five LOC) in each

of the subject systems. The parameters settings of NiCad used

in our study are mentioned in Table II. With these settings,

NiCad separately detect clones and vulnerabilities, distinguish the vulnerabilities located in non-cloned portion of code, and compute all the metrics described in Section II. To verify the statistical significance of the results derived from our quantitative analysis, we also apply the statistical Mann-Whitney-Wilcoxon (MWW) test [1] with \( \alpha = 0.05 \). The non-parametric MWW test does not require normal distribution of data, and thus it suits well for our purpose.

Density of a particular vulnerability \( v \) in non-cloned code, denoted as \( d_v(v) \), is calculated by dividing the number of instances of \( v \) found in non-cloned code by the total number of LOC in all non-cloned blocks of code. Mathematically,

\[
d_v(v) = \frac{v_v}{\ell_v}
\]

where \( v_v \) denotes the number of instances of vulnerability \( v \) found in non-cloned code and \( \ell_v \) denotes the total number of LOC in all non-cloned blocks of 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 97 software systems of the Qualitas Corpus [30], which is a large curated collection of open source systems of diverse application domains and written in Java.

B. Clone Detection

Using the NiCad [28] clone detector (version 3.5), we separately detect code clones (with at least five LOC) in each of the subject systems. The parameters settings of NiCad used in our study are mentioned in Table II. 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 [28]. Then, we compute the pure Type-2 and pure Type-3 clones in accordance with their specifications outlined in Section II.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LawOfDemeter (LD)</td>
<td>Program unit needing too much knowledge about other units.</td>
</tr>
<tr>
<td>LocalVariableCouldBeFinal (LVF)</td>
<td>Local variable assigned only once but not declared final.</td>
</tr>
<tr>
<td>ShortVariable (SV)</td>
<td>A field, local, or parameter with a too short name.</td>
</tr>
<tr>
<td>OnlyOneReturn (OR)</td>
<td>Method with more than one exit points.</td>
</tr>
<tr>
<td>IfStmtsMustUseBraces (ISB)</td>
<td>’if’ statements without accompanying curly braces.</td>
</tr>
<tr>
<td>AssertionsShouldIncludeMessage (AIM)</td>
<td>Assertions including no error message.</td>
</tr>
<tr>
<td>UselessParentheses (UP)</td>
<td>Useless parentheses in code.</td>
</tr>
<tr>
<td>IfElseStmtsMustUseBraces (IEB)</td>
<td>’if-else’ statements without accompanying curly braces.</td>
</tr>
<tr>
<td>AvoidInstantiatingObjectsInLoops (AOL)</td>
<td>Instantiation of new objects inside loop.</td>
</tr>
<tr>
<td>NullAssignment (NA)</td>
<td>Assignment of a “null” to a variable (outside of its declaration).</td>
</tr>
<tr>
<td>ConfusingTernary (CT)</td>
<td>Use of negation within an ‘if’ expression in ‘if-else’ statement.</td>
</tr>
<tr>
<td>AvoidLiteralInIfCondition (ALC)</td>
<td>Use of hard coded literals in conditional statements.</td>
</tr>
<tr>
<td>MethodShouldUseAnnotation (UA)</td>
<td>Missing annotations for methods.</td>
</tr>
<tr>
<td>DataflowAnomaly (DA)</td>
<td>Local definitions and references to variables on different paths.</td>
</tr>
<tr>
<td>ModifiedCyclomaticComplexity (MCC)</td>
<td>A variant of Cyclomatic complexity, which treats switch statements as a single decision point.</td>
</tr>
<tr>
<td>TooManyMethods (TMM)</td>
<td>A class with too many methods.</td>
</tr>
<tr>
<td>NPathComplexity (NPC)</td>
<td>Too high number of acyclic execution paths through a method.</td>
</tr>
<tr>
<td>AvoidCatchingGenericException (ACE)</td>
<td>Use of higher level exception in catching low level error conditions.</td>
</tr>
<tr>
<td>CommentRequired (CR)</td>
<td>Missing required comment for specific language elements.</td>
</tr>
<tr>
<td>MethodArgumentCouldBeFinal (MAF)</td>
<td>Non-final method argument that is never assigned to.</td>
</tr>
<tr>
<td>CommentSize (CS)</td>
<td>Dimensions of non-header comments exceeding specified limits.</td>
</tr>
<tr>
<td>BeanMembersShouldSerialize (BMS)</td>
<td>Class’s member variables not marked as transient, static, or missing accessor methods.</td>
</tr>
<tr>
<td>VariableNamingConventions (VNC)</td>
<td>Named of final variables not fully capitalized or use of underscores in names of non-final variables.</td>
</tr>
<tr>
<td>LongVariable (LV)</td>
<td>Too long name for a field, method or local variable.</td>
</tr>
<tr>
<td>FieldDeclarationsShouldBeAllStartOfClass (FDC)</td>
<td>Class’s member fields not declared at the top of the class.</td>
</tr>
<tr>
<td>DefaultPackage (DP)</td>
<td>Use of default package private accessibility instead of explicit scoping.</td>
</tr>
<tr>
<td>UnusedModifier (UM)</td>
<td>Use modifiers in such a place of code which will be ignored by compiler.</td>
</tr>
<tr>
<td>RedundantFieldInitializer (RFI)</td>
<td>Unnecessary explicit initialization of class’s member fields.</td>
</tr>
<tr>
<td>ImmutableField (IMF)</td>
<td>Class’s private fields whose values never change once they are initialized but not made final.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clone Types</th>
<th>NiCad Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-1</td>
<td>Dissimilarity Threshold</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>Identifier Renaming</td>
<td>No Rename</td>
</tr>
<tr>
<td>Type-2</td>
<td>Dissimilarity Threshold</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>Identifier Renaming</td>
<td>Blind Rename</td>
</tr>
<tr>
<td>Type-3</td>
<td>Dissimilarity Threshold</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>Identifier Renaming</td>
<td>No Rename</td>
</tr>
</tbody>
</table>

C. Vulnerability Detection

For the detection of vulnerabilities in source code, we use PMD (version 5.3.2) [23], which applies a static rule-

based approach for source code analysis and identification of potential vulnerabilities in a software system. For vulnerability detection, we execute PMD from command line interface, and feed to it a set of rules, which is the default rule-set packaged with the Eclipse plugin variant of the tool. All others parameters of PMD are set to the defaults. Using PMD, we separately detect vulnerabilities in each of the subject systems in our study.

IV. Analysis and Findings

Upon detection of the clones and vulnerabilities, 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. To verify the statistical significance of the results derived from our quantitative analysis, we also apply the statistical Mann-Whitney-Wilcoxon (MWW) test [1] with \( \alpha = 0.05 \). The non-parametric MWW test does not require normal distribution of data, and thus it suits well for our purpose.

TABLE I

**ABRIDGED DESCRIPTION OF MAJOR VULNERABILITIES FOUND IN THE SUBJECT SYSTEMS**

TABLE II

**NICAD SETTINGS FOR CODE CLONE DETECTION**
A. Comparative Vulnerability of Cloned vs. Non-Cloned Code

Figure 2 presents how the total number of vulnerabilities are distributed in non-cloned code and different types of clones over all the systems. As seen in Figure 2, 77% of all vulnerabilities are found in non-cloned source code, whereas the clones contain only 23% of vulnerabilities.

The box-plot in Figure 4 presents the densities of vulnerabilities w.r.t. BOC (computed using Equation 1, Equation 2, and Equation 3) 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 4, the density of vulnerabilities (w.r.t. BOC) in non-cloned blocks is much higher than that in clones.

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 and Figure 4. 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 Figure 3. Notice that the distribution of vulnerabilities (Figure 2) is very similar to the distribution of LOC (Figure 3) in non-cloned code and different types of clones. The average LOC in cloned and non-cloned blocks over all the systems are found to be 8.43 and 19.03 respectively. In the subject systems used in our study, 74% of the source code are clone-free over all the systems as portrayed in Figure 3.

To verify this possibility, we compute the distribution of LOC in non-cloned code and different types of clones over all the subject systems. Figure 5 shows that the densities of vulnerabilities (w.r.t. LOC) in cloned and non-cloned code are almost equal (with a mean difference of 0.02 only). Thus, it appears that there is no significant difference in the density of vulnerabilities w.r.t. LOC in cloned versus non-cloned code. A MWW test ($P = 0.28, P > \alpha$) over distribution of densities of vulnerabilities (w.r.t. LOC) across all the subject systems also confirms this finding. Therefore, we derive the answer to the RQ1 as follows:

 Ans. to RQ1: Cloned code are NOT more vulnerable than non-cloned code. Rather, higher number of vulnerabilities can be found in non-cloned code due to their larger sizes (in terms of LOC) as compared to cloned code.
B. Comparative Vulnerability of Different Types of Clones

The distribution of vulnerabilities portrayed in Figure 2 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 slightly 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 seen in Figure 3, which might be a reason why a higher number of vulnerabilities are found in clones of this particular category.

The box-plot in Figure 4 indicates that density of vulnerabilities w.r.t. BOC is higher in Type-1 clones as compared to pure Type-2 and pure Type-3. MWW tests between the distributions of vulnerabilities (w.r.t. BOC) in each two of the three categories of clones also suggest statistical significance in the differences except for the case of vulnerabilities in pure Type-2 and pure Type-3 clones. The results of the MWW tests are presented in Table III.

<table>
<thead>
<tr>
<th>Clone Types</th>
<th>Type-1</th>
<th>Pure Type-2</th>
<th>Pure Type-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Pure Type-2</td>
<td>$P = 0$</td>
<td>-</td>
<td>$P = 0.7641$</td>
</tr>
<tr>
<td>Pure Type-3</td>
<td>$P = 0.0$</td>
<td>$P = 0.7641$</td>
<td>-</td>
</tr>
</tbody>
</table>

In Figure 5, the differences in the densities of vulnerabilities w.r.t. LOC in the three categories of clones are relatively higher, and the density is the highest in Type-1 clones while lowest in pure Type-3. MWW tests between the distributions of vulnerabilities (w.r.t. LOC) in each pair of the three categories of clones also suggest statistical significance in the differences. The results of the MWW tests are presented in Table IV.

<table>
<thead>
<tr>
<th>Clone Types</th>
<th>Type-1</th>
<th>Pure Type-2</th>
<th>Pure Type-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Pure Type-2</td>
<td>$P = 0.0173$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Pure Type-3</td>
<td>$P = 0.0$</td>
<td>$P = 0.0$</td>
<td>-</td>
</tr>
</tbody>
</table>

Based on the findings, we now answer the RQ2 as follows:

**Ans. to RQ2:** Although the clones larger in size (w.r.t LOC) are more vulnerable than smaller clones, in general, Type-1 clones are the most vulnerable while pure Type-3 clones are the least vulnerable and pure Type-2 clones fit in between.

C. Relatively Frequent Vulnerabilities

To address the third research question (i.e., RQ3), we compute the densities of each individual vulnerability separately in cloned and non-cloned code (over all the subject systems) using Equation 7 and Equation 8 respectively. Then we distinguish 20 vulnerabilities, which have the highest densities in cloned code over all the subject systems. Let $D_c$ denote the set of these 20 vulnerabilities. Similarly, we form another set $D_{nc}$ consisting of 20 vulnerabilities having the highest densities in non-cloned code. By the union of these two sets we obtain a set $D$ of 29 vulnerabilities that have the highest densities across both cloned and non-cloned code. Mathematically, $D = D_c \cup D_{nc}$.

Short descriptions of these 29 vulnerabilities are given in Table I; further elaborations can be found in [23]. The densities of each of these 29 vulnerabilities in cloned and non-cloned code are presented in Table V. Note that, these 29 vulnerabilities represent 85% of total vulnerabilities in cloned code and 89% of the vulnerabilities found in non-cloned code over all the subject systems. Next, we want to partition these vulnerabilities into three clusters: one with vulnerabilities dominating in cloned code, another with those dominating in non-cloned code and a third cluster with vulnerabilities, which almost equally appear in both cloned and non-cloned code.

**Cluster Analysis:** For the purpose of aforementioned partitioning, we conduct a clustering analysis on the densities of these vulnerabilities in cloned and non-cloned code. For each $v$ of these 29 vulnerabilities, we compute a ratio $\mathcal{M}(v)$ as follows:

$$\mathcal{M}(v) = \frac{d_c(v)}{d_{nc}(v)}, \text{ where } v \in D$$ (9)

The ratios computed for each of the 29 vulnerabilities are presented in the second column from the right in Table V. Notice that, for a particular vulnerability $v$, the ratio $\mathcal{M}(v)$ close to 1.0 indicates that the vulnerability $v$ almost equally appears in both cloned and non-cloned code. If $\mathcal{M}(v)$ is much higher than 1.0, the appearance of vulnerability $v$ can be characterized to have dominated in cloned code. Similarly, $\mathcal{M}(v)$ being much lower than 1.0 implies that the vulnerability $v$ appears more in non-cloned code. However, a threshold scheme seems required to determine when the value of $\mathcal{M}(v)$ can be considered significantly close to or distant from 1.0.
Instead of setting an arbitrary threshold by ourselves, we apply unsupervised Hierarchical Agglomerative Clustering [6] for partitioning the values of $M(v)$. The dendrogram produced from this statistical clustering is presented in Figure 6. In the dendrogram, three major clusters are evident, two marked with dotted rectangles and the third left unmarked in the middle. The values of $M(v)$ for the vulnerabilities in the middle cluster range between 0.97 and 1.92. This middle cluster includes a set of those vulnerabilities, which equally appear in both cloned and non-cloned codes. Let $G_b$ denote this cluster.

For all of the five vulnerabilities (i.e., MCC, NPC, DA, ACE, and TMM) in the right-most cluster $M(v) \geq 2.38$, which indicates that these vulnerabilities appear more frequently in cloned code compared to their presence in non-cloned code. Let $G_c$ denote the cluster of these vulnerabilities. The leftmost cluster, denoted as, $G_a$, includes the vulnerabilities with $M(v) < 0.97$, and they are frequently found in non-cloned code. The right-most column in Table V labels the vulnerabilities in accordance with how they are clustered here.

**Statistical Significance:** For each of the three clusters of vulnerabilities, we separately conduct MWW tests between the densities of those vulnerabilities in cloned and non-cloned code to determine the statistical significance of the difference in their existence in those two categories (i.e., cloned and non-cloned) of code. The results of the separate MWW tests over each of the clusters are presented in Table VI.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>$G_a$</th>
<th>$G_c$</th>
<th>$G_b$</th>
<th>$P$-values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0474</td>
<td>0.0024</td>
<td>0.3575</td>
<td>0.0024</td>
</tr>
</tbody>
</table>

The $P$-values in Table VI indicate statistical significance in the differences of density-distribution in cloned and non-cloned code for vulnerabilities in cluster $G_a$ and $G_c$, but not for those in cluster $G_b$. Thus, our clustering of the vulnerabilities is confirmed accurate with statistical significance. Now, we answer the research questions RQ3 as follows:

**Ans. to RQ3:** There are distinct sets of vulnerabilities (as characterized in Table V), which frequently appear in cloned code or non-cloned code, while many other vulnerabilities are found to be equally present in both cloned and non-cloned code.

**V. Threats to Validity**

In this section, we discuss possible threats to the validity of our study and how we have mitigated their effects.

**Construct Validity:** In the detection of vulnerabilities with PMD, we used its default settings and relied on the set of rules, which came with the Eclipse plug-in variant of the tool. Those set of rules might not have covered all possible vulnerabilities, and we considered each vulnerability equally important. In the selection of the two dominant sets of vulnerabilities (Section IV-C), we picked top 20 vulnerabilities for each set having the highest densities in cloned and non-cloned code. Although those chosen vulnerabilities cover more than 80% of all distinct vulnerabilities and 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.

**Internal Validity:** The clone detector, NiCad, used in our study, is reported to be very accurate in clone detection [28], and we have carefully set NiCad’s parameters. The tool, PMD, used in our work for vulnerability detection, is also known effective and widely used in both industry and research community. However, 15 out of 112 systems in the Quality Corpus [30] were failed to be processed by either NiCad or PMD. Those 15 systems are excluded from our study. Moreover, we 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 Java. Thus the findings from this work may not be generalizable for industrial systems and source code written in languages other than Java.

**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 NiCad and PMD are also available online. Therefore, it should be possible to replicate the study.

**VI. RELATED WORK**

It is often believed that inconsistent changes to clones cause program faults and frequent changes may lead to significant instances of inconsistent changes [3], [14]. Thus, to develop an understanding on the fault-proneness of code clones, studies have been conducted to examine the stability (in terms of frequency and sizes of changes) and inconsistent changes in evolving code clones.

Juergens et al. [14] reported that inconsistent changes to clones are very frequent and a significant number of faults are induced by such inconsistent changes. Barbour et al. [3] suggested that late propagations due to inconsistent changes are prone to introduce software defects. While Lozano and Wermeling [20] suggested that having a clone may increase the maintenance effort for changing a method, Hotta et al. [10] reported code clones not to have any negative impact on software changeability. Lozano et al. [21] reported that a vast majority of methods experience larger and frequent changes when they contain cloned code. Mondal et al. [22] also reported code clones to be less stable. However, opposite results are found from the other studies [2], [8], [9], [17].
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. [19], who reported that only 4% of the bugs were found in duplicated code. In a similar study, Rahman et al. [24] also observed that majority of bugs were not significantly associated with clones. These findings contradict with those of Juergens et al. [14] and Barbour et al. [3].

The contradictory results from the earlier studies imply 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 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 unique aspect of our work.

VII. CONCLUSION

In this paper, we have presented a quantitative empirical study on the vulnerabilities (in terms of bad coding patterns) in different types of code clones and non-cloned code in 97 open-source software systems written in Java. To the best of our knowledge, no other work in the past conducted a comparative study of such vulnerabilities in cloned and non-cloned as done in our work. In our study, we have found no significant differences between the densities of vulnerabilities in code clones and clone-free source code. Surprisingly, among the three categories (i.e., Type-1, pure Type-2, and pure Type-3) of clones studied in our work, Type-1 clones are found to be the most vulnerable whereas pure Type-3 are the least. In addition, our study identifies a set of five vulnerabilities that appear more frequently in cloned code compared to non-cloned code. Another set of 11 vulnerabilities are also distinguished, which are more frequently found in non-cloned code as opposed to cloned code. The results are validated in the light of statistical significance.

The findings from this study significantly advance our understanding of the characteristics, impacts, and implications of code clones in software systems. These findings can 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. For example, since Type-1 clones are found to be the most vulnerable, and that refactoring of Type-1 clones can be expected to be easier (due to absence of much differences among them) than refactoring other types of clones, we argue that this particular type of clones should be removed from source code by frequent refactoring.

Acknowledgement: Thanks to Pradeep Jakibanjar and Rahul Chatterjee for helping in data collection and analysis respectively.

REFERENCES