2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

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.

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.90

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-cloned 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 category x clones, denoted as ∂_x^β, is defined as the ratio of the number of vulnerabilities found in clones of category x and the number of clones of category x where, category $x \in \{T_1, T_2, T_3, T_2^p, T_3^p\}$. Mathematically,

$$\partial_x^\beta = \frac{\nu_x}{\beta_x} \tag{1}$$

where ν_x denotes the number of vulnerabilities found in clones of category x and β_x denotes the total number of clones of category x.

Density of vulnerabilities w.r.t. BOC in all clones, denoted as ∂_c^β, is defined as the ratio of the number of vulnerabilities found in all clones and total the number of all the clones. Mathematically,

$$\partial_c^\beta = \frac{\nu_c}{\beta_c} \tag{2}$$

where ν_c denotes the number of vulnerabilities found in all clones and β_c denotes the total number of all categories of clones.

Density of vulnerabilities w.r.t. BOC in non-cloned code, denoted as $\partial_{\bar{c}}^\beta$, 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,

$$\partial_{\bar{c}}^\beta = \frac{\nu_{\bar{c}}}{\beta_{\bar{c}}} \tag{3}$$

where $\nu_{\bar{c}}$ denotes the number of vulnerabilities found in non-cloned code and $\beta_{\bar{c}}$ denotes the total number of non-cloned blocks of code.

Density of vulnerabilities w.r.t. LOC in category x clones, denoted as ∂_x^ℓ, 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,

$$\partial_x^\ell = \frac{\nu_x}{\ell_x} \tag{4}$$

where ν_x denotes the number of vulnerabilities found in clones of category x and ℓ_x denotes the total number of LOC in clones of category x.

Density of vulnerabilities w.r.t. LOC in all clones, denoted as ∂_c^ℓ, is defined as the ratio of the number of vulnerabilities found in all clones and the number of LOC in all the clones. Mathematically,

$$\partial_c^\ell = \frac{\nu_c}{\ell_c} \tag{5}$$

where ν_c denotes the number of vulnerabilities found in all clones and ℓ_c denotes the total number of LOC in all clones.

Density of vulnerabilities w.r.t. LOC in non-cloned code, denoted as $\partial_{\bar{c}}^\ell$, 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,

$$\partial_{\bar{c}}^\ell = \frac{\nu_{\bar{c}}}{\ell_{\bar{c}}} \tag{6}$$

where $\nu_{\bar{c}}$ denotes the number of vulnerabilities found in non-cloned code and $\ell_{\bar{c}}$ 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,

$$d_c(v) = \frac{v_c}{\ell_c} \tag{7}$$

where v_c denotes the number of instances of vulnerability v found in cloned code and ℓ_c denotes the total number of LOC across all clones.

TABLE I
ABRIDGED DESCRIPTION OF MAJOR VULNERABILITIES FOUND IN THE SUBJECT SYSTEMS

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

Density of a particular vulnerability v in non-cloned code, denoted as $d_{\bar{c}}(v)$, is calculated by dividing the number of instances of v found in non-cloned code by the total number of LOC in non-cloned blocks of code. Mathematically,

$$d_{\bar{c}}(v) = \frac{v_{\bar{c}}}{\ell_{\bar{c}}} \tag{8}$$

where $v_{\bar{c}}$ denotes the number of instances of vulnerability v found in non-cloned code and $\ell_{\bar{c}}$ 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 II
NICAD SETTINGS FOR CODE CLONE DETECTION

Clone Types	`NiCad` Parameter	Value
Type-1	Dissimilarity Threshold	0%
	Identifier Renaming	No Rename
Type-2	Dissimilarity Threshold	0%
	Identifier Renaming	Blind Rename
Type-3	Dissimilarity Threshold	30%
	Identifier Renaming	No Rename

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.

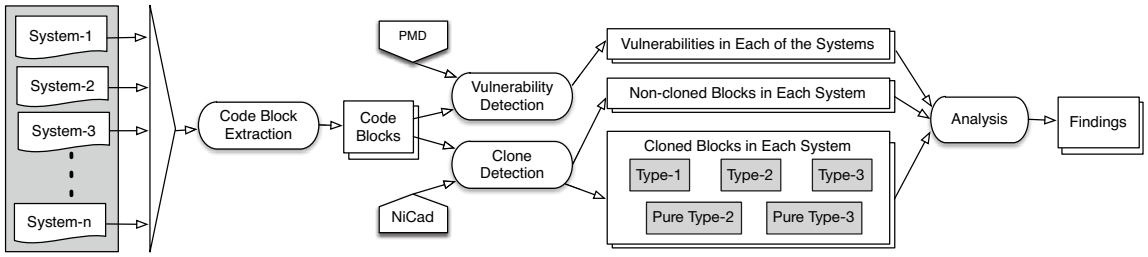

Fig. 1. Procedural Steps of the Empirical Study

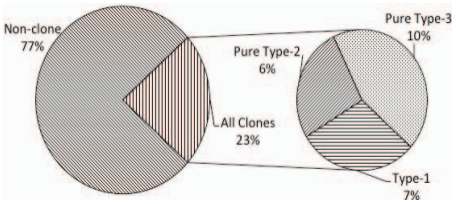

Fig. 2. Distribution of Vulnerabilities in Cloned and Non-cloned Code

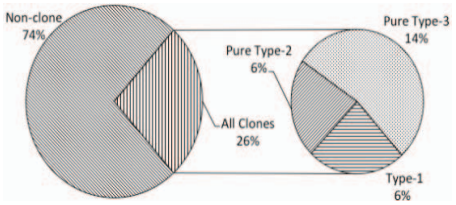

Fig. 3. Distribution of LOC in Cloned and Non-cloned Code

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,

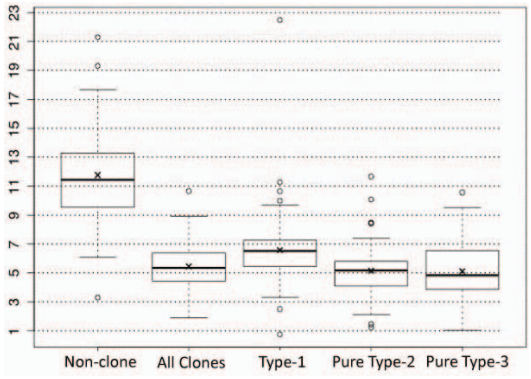

Fig. 4. Densities of Vulnerabilities w.r.t. BOC

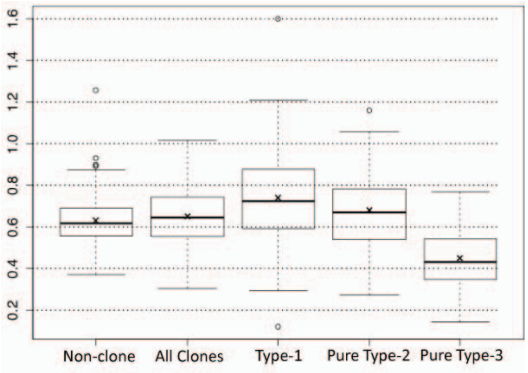

Fig. 5. Density of Vulnerabilities w.r.t. LOC

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. Thus, the possibility of influence of code size (in terms of LOC) on the number or density of vulnerabilities w.r.t. *BOC* is found to be true.

We, therefore, perform a deeper investigation 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 4, Equation 5, and Equation 6) found in non-cloned code and in 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 III
MWW TESTS OVER DENSITIES OF VULNERABILITIES W.R.T. *BOC* IN DIFFERENT CATEGORIES OF CLONES

Clone Types	*Type-1*	*Pure Type-2*	*Pure Type-3*
Type-1	-	$P = 0.0$	$P = 0.0$
Pure Type-2	$P = 0.0$	-	$P = 0.7641$
Pure Type-3	$P = 0.0$	$P = 0.7641$	-

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 IV
MWW TESTS OVER DENSITIES OF VULNERABILITIES W.R.T. *LOC* IN DIFFERENT CATEGORIES OF CLONES

Clone Types	*Type-1*	*Pure Type-2*	*Pure Type-3*
Type-1	-	$P = 0.0173$	$P = 0.0$
Pure Type-2	$P = 0.0173$	-	$P = 0.0$
Pure Type-3	$P = 0.0$	$P = 0.0$	-

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 $\mathcal{D}_c$ denote the set of these 20 vulnerabilities. Similarly, we form another set $\mathcal{D}_{\bar{c}}$ consisting of 20 vulnerabilities having the highest densities in non-cloned code. By the union of these two sets we obtain a set $\mathcal{D}$ of 29 vulnerabilities that have the highest densities across both cloned and non-cloned code. Mathematically, $\mathcal{D} = \mathcal{D}_c \cup \mathcal{D}_{\bar{c}}$.

TABLE V
VULNERABILITIES DOMINATING IN CLONED AND NON-CLONED CODE

Vulner-ability (v)	**Density** **Clone** $d_c(v)$	**Non-clone** $d_{\bar{c}}(v)$	**Ratio** $\mathcal{M}(v) = \frac{d_c(v)}{d_{\bar{c}}(v)}$	**High Frequency Area**
LD	0.1489	0.10547	1.4118	**Both** clone and non-clone code
LVF	0.08561	0.05885	1.4547	
SV	0.02258	0.02149	1.0507	
OOR	0.03129	0.01625	1.9255	
ISB	0.02124	0.01605	1.3234	
AIM	0.00968	0.00707	1.3692	
UP	0.00822	0.00602	1.3654	
IEB	0.00472	0.00485	0.9732	
AOL	0.00428	0.00301	1.4219	
NA	0.00423	0.00258	1.6395	
CT	0.00417	0.00239	1.7448	
ALC	0.00406	0.00274	1.4818	
UA	0.00389	0.00218	1.7844	
DA	0.0563	0.02362	2.3836	**Clone** code only
MCC	0.01026	0.0013	7.8923	
TMM	0.00421	0.0002	21.0500	
NPC	0.00398	0.00087	4.5747	
ACE	0.00389	0.00127	3.0630	
CR	0.03638	0.08105	0.4489	**Non-clone** code only
MAF	0.04668	0.08102	0.5762	
CS	0.00227	0.04542	0.0500	
BMS	0.00006	0.02532	0.0024	
VNC	0.00277	0.01866	0.1484	
LV	0.00309	0.01727	0.1789	
FDC	0.00002	0.00872	0.0023	
DP	0.0033	0.00831	0.3971	
UM	0.00002	0.00517	0.0039	
RFI	0.00004	0.00467	0.0086	
IMF	0.00001	0.00444	0.0023	

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:

$$\mathcal{M}(v) = \frac{d_c(v)}{d_{\bar{c}}(v)},\ where,\ v \in \mathcal{D} \tag{9}$$

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.

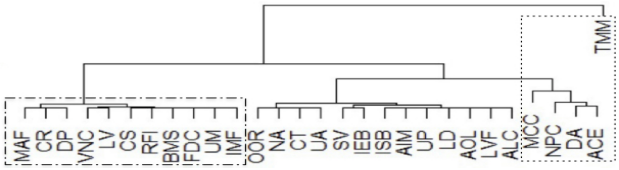

Fig. 6. Hierarchical Agglomerative Clustering of Vulnerabilities

Instead of setting an arbitrary threshold by ourselves, we apply unsupervised *Hierarchical Agglomerative Clustering* [6] for partitioning the values of $\mathcal{M}(v)$. 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 $\mathcal{M}(v)$ 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 $\mathcal{M}(v) \geq 2.38$, which indicates that these vulnerabilities appear more frequently in *cloned* code compared to their presence in non-cloned clone. Let G_c denote the cluster of these vulnerabilities. The left most cluster, denoted as, $G_{\bar{c}}$, includes the vulnerabilities with $\mathcal{M}(v) < 0.97$, 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 VI
MWW TESTS BETWEEN DENSITY-DISTRIBUTIONS IN CLONED AND NON-CLONED CODE FOR VULNERABILITIES OF EACH CLUSTER

Cluster	G_c	$G_{\bar{c}}$	G_b
P**-values**	0.0474	0.0024	0.3575

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_c and $G_{\bar{c}}$, 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 question *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 Qualitus 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 Wermelinger [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

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

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

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

[4] I. Baxter, M. Conradt, J. Cordy, and R. Koschke. Software clone management towards industrial application (dagstuhl seminar 12071). *DagStuhl Report*, 2(2):21–57, 2012.

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

[6] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. A Wiley-Interscience Publication, 2000.

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

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

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

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

[11] 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.

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

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

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

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

[16] C. Kapser and M. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, 2008.

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

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

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

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

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

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

[23] PMD. *PMD - Source Code Analyzer*. http://pmd.sourceforge.net, last access: Dec 2015.

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

[25] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, Institut fuï Informatik und angewandte Mathematik, Germany, 2005.

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

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

[28] 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.

[29] 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.

[30] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *APSEC*, pages 336 – 345, 2010.

[31] 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.

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

[33] 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.