JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2009; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Near-miss function clones in open source software: an empirical study

C. K. Roy*,† and J. R. Cordy‡

School of Computing, Queen's University, Kingston, ON, Canada K7L 3N6

SUMMARY

The new hybrid clone detection tool NICAD combines the strengths and overcomes the limitations of both text-based and AST-based clone detection techniques and exploits novel applications of a source transformation system to yield highly accurate identification of cloned code in software systems. In this paper, we present an in-depth study of near-miss function clones in open source software using NICAD. We examine more than twenty open source C, Java and C# systems, including the entire Linux Kernel, Apache httpd, J2SDK-Swing and db4o and compare their use of cloned code in several different dimensions, including language, clone size, clone similarity, clone location and clone density both by proportion of cloned functions and lines of cloned code. We manually verify all detected clones and provide a complete catalogue of different clones in an online repository in a variety of formats. These validated results can be used as a cloning reference for these systems and as a benchmark for evaluating other clone detection tools.

KEY WORDS: Near-miss Function Clones; Empirical Study; Open Source Software

1. Introduction

Reusing a code fragment by copying and pasting with or without minor modifications is a technique frequently used by programmers, and thus software systems often have duplicate fragments of code in them. Such duplicated fragments are called *code clones* or simply *clones*.

Although cloning is beneficial in some cases [18, 5] and often programmers intentionally use it [22], it can be detrimental to software maintenance [11, 16]. For example, if a bug is detected in a code fragment, all the fragments similar to it should be investigated to check for the same bug [24], and when enhancing or adapting a piece of code, duplicated fragments can multiply the work to be done [16, 26]. A recent study that works on industrial code shows that inconsistent changes to code duplicates are frequent and lead to severe unexpected behavior [17].

*Correspondence to: School of Computing, Queen's University, Kingston, ON, Canada K7L 3N6
†E-mail: croy@cs.queensu.ca
‡E-mail: cordy@cs.queensu.ca

Received 23 February 2009
Revised X X 2009

In response, over the past decade several techniques and tools for detecting code clones have been proposed [34, 32, 33]. However, despite a decade of active research there has been a marked lack of in-depth comparative studies of cloning, particularly in a variety of systems. There have been many empirical studies on cloning, for example every new technique comes with some sort of empirical validation [32], and empirical studies are used when comparing tools [6, 7, 10, 8, 37, 38].

However, in both cases the focus is on validating or comparing the techniques rather than the clone properties of the subject systems themselves. Particular subject systems have also been analyzed with respect to aspects such as harmfulness/usefulness and maintenance issues of clones [18, 5, 22, 23], taxonomies of clones [20, 19] or evolution of clones [4], and there has been one in-depth study [27] on exact clones in web applications.

In this paper we provide an in-depth empirical study of function clones in more than twenty open source C, Java and C# systems including the entire Linux Kernel, Apache *httpd*, 2SDK-Swing and db4o, using our recently introduced hybrid approach, NICAD [31], which combines the strengths and overcomes the limitations of both text-based and AST-based techniques and exploits novel applications of a source transformation system.

NICAD has been found to be effective in detecting near-miss function clones [31]. However, it was applied to only two small C systems, focusing on its efficacy in detecting copy/pasted near-miss clones by using flexible pretty-printing, code normalization and code filtering. In this paper, we exploit further improvements to NICAD to deal with large systems, using a dynamic clustering technique and distributing the comparison load across multiple processors, and on its use with Java and C# systems.

This paper is an extended version of our WCRE'08 paper [30], in which we provided a concise in-depth study of clones in open source C and Java systems. Here we further study six open source C# systems of various sizes, and compare the cloning characteristics of these with those in C and Java, as well as comparing all of the languages and systems with respect to clone localization across subsystems. In particular, we focus on three specific research questions:

(1) *What is the cloning status of open source systems? Are there many clones? Are there more near-miss clones than exact clones?* (2) *Are there significant differences in cloning between the different language paradigms used in open source systems?* (3) *Are there significant differences in cloning in large systems compared to medium and small-sized open source systems?*

We manually verify all detected clones and provide a complete catalogue of different clones in an online repository [29] in a variety of formats. These validated results can be used as a cloning reference for these systems and as a benchmark for evaluating other clone detection tools.

Although NICAD is designed to allow for flexible pretty-printing, code normalization and filtering, in this paper we focus on detecting only exact and near-miss function clones, using only the basic NICAD technique, consisting of standard pretty-printing of code fragments to encode structure, followed by text line comparison at a variety of dissimilarity thresholds. Our study demonstrates that NICAD is capable of detecting clones in very large systems in many different languages, and that there is a significant proportion of code in these systems that has been reused by copy/paste.

The rest of the paper is organized as follows. Following a short introduction to NICAD in Section 2, we provide the experimental setup of our study in Section 3. Section 4 presents and analyzes our findings, and Section 5 considers other empirical studies and their relation to ours. Finally, Section 6 concludes the paper with directions of future research.

2. NICAD Overview

The clone detector used in this study is a significantly improved and adapted version of NICAD [31], which works in three phases *Extraction*, in which all potential clones (code fragments of the target kind) are identified and extracted, *Comparison*, in which the potential clones are clustered and compared, and *Reporting*, in which discovered clone pairs and classes are related to original source and presented for human inspection.

NICAD begins by extracting the set of all code fragments of the desired granularity in the system, each to a set of pretty-printed and source-coordinate annotated *potential clones*. Pretty-printing allows us to use efficient text comparison while remaining structure sensitive by preserving the parsed structure in the pretty-printed code.

In this paper we are interested in function clones, so NICAD was asked to extract all function and method definitions with their original source-coordinates. Because we are interested in intentional copy/pasted clones, C macros are not expanded, but *#ifdefs* are resolved using the method of Antoniol et al. [4] to comment out all *#else* parts.

NICAD compares the pretty-printed potential clones line-by-line textwise using a longest common subsequence (LCS) algorithm similar to the Unix *diff* utility. To determine whether two potential clones really are clones of each other, we compare their pretty-printed (and optionally normalized/filtered) text lines and use the number of unique lines in each as a measure of similarity/dissimilarity. In particular, we compute the size-sensitive Unique Percentage of Items (UPI) for each potential clone using the equation (for details see [31]):

$$UPI = \frac{No.\ of\ Unique\ Items * 100}{Total\ No.\ of\ Items}$$

If the UPI for both line sequences is zero or below a certain predefined UPI threshold (UPIT), the potential clones are considered to be clones. The number of comparisons is optimized using an exemplar-based technique that builds clone classes directly by choosing the largest unclassified potential clone as an exemplar and comparing similarly-sized potential clones to it depending on the UPI. Results from NICAD are reported in both XML database form and as an interactive HTML website to assist hand validation.

3. Experimental Setup

In this experiment we have applied NICAD to find function clones in a number of open source systems. We have then used a set of metrics to analyze the results. This section introduces the systems we have studied and the metrics used, including a brief overview of our definition and methodology for manual verification of the detected clones.

3.1. Subject Systems

In this study we have analyzed ten C, seven Java and six C# systems varying in size from 4K LOC to 6265K lines of code (LOC) and including the entire Linux Kernel. In Table I we provide a statistical overview of these subject systems (only *.c, .java* and *.cs* files were considered in the calculations).

Because Bellon's experiment [6] is the most extensive to date, we have chosen all the C and Java systems of his experiment including the systems used in his test run. In addition, we have studied

Table I. Overview of the subject systems

Language	Subject System	No. of Files	Lines of Code	No. of Methods
C	Abyss [1]	10	4K	148
	Bison [6]	57	16K	315
	Cook [6]	287	70K	1362
	Gzip-1.2.4 [12]	22	8K	117
	Apache-httpd-2.2.8 [2]	496	275K	4301
	Postgresql [6]	314	202K	4669
	snns [6]	138	94K	2201
	Weltab [6]	39	11K	123
	Wget [6]	23	17K	219
	Linux-2.6.24.2 [21]	9491	6265K	154977
Java	Eclipse-ant [6]	161	35K	1754
	EIRC [6]	54	11K	588
	Netbeans-Javadoc [6]	97	14K	972
	Eclipse-jdtcore [6]	582	148K	7383
	JHotDraw 5.4b1 [15]	233	40K	2399
	Spule [6]	50	13K	420
	j2sdk-swing [6]	414	204K	10971
C#	VMukti-Chat [39]	31	3K	73
	Linq [39]	71	17K	638
	nant-0.86 [39]	438	105K	2383
	RssBandit-1.5.0.17 [39]	316	167K	4587
	Castle [39]	2419	270K	9530
	db4o-7.4 [39]	2220	235K	13855

Apache *httpd* [2], *JHotDraw* [15], the entire Linux Kernel [21] and a number of smaller systems. We have also added six C# systems of different sizes and kinds, including *db4o*, a popular and widely used production commercial and open source object database. Although it is partly adapted from its Java version, it differs significantly from the original and is the largest C# system ever studied for clones.

Since the Linux kernel is almost two orders of magnitude larger than any of the other systems, we have treated it as an outlier. We provide statistical results both including and not including Linux in Section 4.1, and in later subsections we have dropped Linux from averages and provided Linux results separately to avoid any bias due to its exceptionally large size.

3.2. Clone Definition

In this study we have considered all non-empty functions of at least 3 LOC in pretty-printed format (function header and opening bracket on the first line, at least one code line, and ending bracket on the third line). Empty functions, which are common in some systems, have intentionally not been considered. We then use different UPI (difference) thresholds [31] to find exact and near-miss (copy/paste/modify) function clones. For example, if the UPI threshold is 0.0, we detect only exact clones; if the UPI threshold 0.10, we detect two functions as clones if at least 90% of the pretty-printed text lines are the same (i.e., if they are at most 10% different – see [31]). In this thesis we present

Fragment 1: httpd/srclib/apr/user/win32/userinfo.c(lines 199-244)

```
APR_DECLARE(apr_status_t) apr_uid_get(apr_uid_t *uid, apr_gid_t *gid,
                          const char *username, apr_pool_t *p)
    {
        #ifdef _WIN32_WCE
            return APR_ENOTIMPL;
        #else
            SID_NAME_USE sidtype;

      (36 more lines)

            return APR_SUCCESS;
        #endif }
```

Fragment 2: httpd/srclib/apr/user/netware/userinfo.c (lines 59-63)

```
APR_DECLARE(apr_status_t) apr_uid_get(apr_uid_t *uid, apr_gid_t *gid,
                          const char *username, apr_pool_t *p)
    {
       return APR_ENOTIMPL; }
```

Figure 1: Example of possible false positive clones

our results for the representative set of UPI thresholds 0.0, 0.10, 0.20 and 0.30, although we have also tested 0.05, 0.15, 0.25 and 0.35 in our work.

3.3. Validation of Clones

All clones detected in this study were validated by hand. To validate detected clones we used a two-step process. First, we used NICAD's interactive HTML web page output to give an overall view of the original source of the clone classes. Second, we used the XML output to pairwise compare the original source of the functions in each clone class using Linux *diff* to determine the textual similarity of the original source. We then manually checked all code clone pairs that had lower similarity values than the UPI threshold chosen. Because of our concise interactive HTML view and tool support for comparing original source, manual validation is not time-consuming, and the total time to manually validate all clones in this experiment was less than one man-month.

We should note that it was not our intention to measure the recall of our tool NICAD in this study, since we did not know the clone status of the systems before we began. However, NICAD has previously been shown to give high recall in both our first experiment [31] and using our mutation-based evaluation framework [36].

In terms of precision, our hand validation of all detected clones did not find any false positives at UPI threshold 0.2 or lower. However, at UPI threshold 0.3 and higher, we noticed that in a few cases our reported clones might be considered false positives, even though they meet our clone definition above. This issue arises specially because of NICAD's handling of preprocessor directives in C systems, which comments out the *#else* part of each *#ifdef*, sometimes leaving many fewer lines, leading to large differences in the original source. However, in such cases it is still not clear whether or not a pair might really be clones, since it is possible that the larger is an *#ifdef* generalization of the other.

For example, the two code fragments (of 45 LOC and 5 LOC respectively) shown in Figure 1 are detected as exact clones in our study because during *if-def* resolution, we comment out the *else* part of the *if-def*. It is not clear whether or not these two are truly clones. By the number of lines and the contents of the *else* part of the first function, they are obviously not clones. On the other hand, it seems quite possible that the bottom fragment was copied and the *if-def* directives were added later on (or vice- versa). However, the number of such cases was very few in our study. We thus say that NICAD gives about 100% precision in this study. Of course, we are using the basic version of NICAD in this study which is without any significant source transformation and that we are dealing with only function clones and therefore, we expect about 100% precision too.

3.4. Metrics and Visualizations

This subsection describes the different metrics and visualizations that we have used in this experiment. These metrics are either adapted or reused from previous studies of cloning [27, 3, 20, 41, 28].

Total Cloned Methods (TCM): In this study we focus on function clones, and thus our first metric is related to the number of methods. By *TCM* we mean the total number of cloned methods in a system for a given UPI threshold (after manual verification). *TCMp* is the percentage *TCM* of the total number of methods in the system. A higher *TCMp* value corresponds to a higher level of method cloning in the system. For example, if the *TCMp* of a system is greater than 50% with UPI threshold 0.0, we can say that the system has more exact cloned methods than non-cloned methods. Such systems have a high update anomaly risk; every update to the system has a greater chance of involving a clone than not.

Since methods can be of different sizes and there may be many clones that are quite small, we also consider similar metrics w.r.t the number of lines in the systems. We define *TCLOC* as the total number of cloned lines of a system for a given UPI threshold and *TCLOCp* as the percentage of total number of lines of the system for a given UPI threshold. Since we apply standard pretty-printing before clone detection, which eliminates formatting and layout differences, resolves *#ifdefs* (in C systems) and ignores comments, we can get an accurate percentage of cloned lines. We thus define the similar metrics w.r.t standard pretty-printed lines of code as *TCSppLOC* and *TCSppLOCp* respectively. In practice, there is not much difference between *TCLOCp* and *TCSppLOCp*, but at the same time *TCSppLOCp* gives a more accurate measure. We thus provide our findings w.r.t *TCSppLOCp* rather than both.

File Associated with Clones (FAWC): While the above metrics provide the overall cloning statistics for a subject system, they cannot provide any clue as to whether the clones are local to some specific files or are scattered all over the system. With *FAWC* we provide these statistics for each system at each UPI threshold. We consider that a file is associated with clones if it has at least one method that forms a clone pair with another method in the same file or a different file. We define *FAWCp* as the percentage of files associated with clones at a given threshold. For example, "*FAWCp* of a system x with UPI threshold 0.0 (exact clones) is 50%" means that 50% of the files of x contain at least one exact cloned method. From a software maintenance point of view, a lower value of *FAWCp* is desirable, since in this case clones are localized to certain specific files and thus may be easier to maintain.

Cloned Ratio of File for Methods (CRFM): While *TCM* related metrics provide a good indication of the overall cloning level and *FAWCp* hints at the overall localization of the clones, still one cannot say which files contain the majority of the clones in the system. With *CRFM* we attempt to discover the highly cloned files. In particular, for a file f, $CRFM(f)$ is defined as follows:

$$CRFM(f) = \frac{Total\ number\ of\ cloned\ methods\ in\ file\ f * 100}{Total\ number\ of\ methods\ in\ file\ f}$$

Where a method is considered to be a *cloned method* if it forms a clone pair/clone class with another method(s) of the same file (e.g., for near-miss clones) or another file (within the same directory or a different directory) and *total number of methods in file f* denotes the number of methods of f that are 3 LOC or more in standard pretty-printed format. Similar metrics are defined w.r.t the lines of code (*CRFLOC*) and standard pretty-printed lines of code (*CRFSppLOC*). These metrics are similar to (but not same as) the *FSA* metric of Rajapakse and Jarzabek [27] and the *RSA* metric of Ueda et al. [41], although they ignore clones that form clone pairs within the same file and we do not.

With *CRFM* we can determine the highly cloned files of a system and possibly can also predict the maintenance difficulty based on the metric values. For example, consider two systems x and y of similar size, both having the same values for the *TCM* related metrics. In x, clones are scattered across the system in such a way that no two files are substantially similar. But in y, clones are well concentrated into a certain set of files. From a clone treatment perspective, system y is more interesting than x because the clones in y might be more easily treatable than those of x.

Qualifying File Count for Methods (QFCM): As in Rajapakse and Jarzabek [27] we define *QFCM(v)* for *CRFM* value v as the number of files for which *CRFM* is not less than v. For example, QFCM(20%) gives the number of files in the system having a *CRFM* value not less than 20%. *QFCMp* is *QFCM* expressed as a percentage of the total number of files in the system. For example, "*QFCMp*(30%+) = 28% for a system x with UPI threshold 0.0" means that 28% of the files of x have 30% or more exact cloned methods. As usual we define similar metrics for source lines of code (*QFCLOC* and *QFCLOCp*) and for pretty-printed lines of code (*QFCSppLOC* and *QFCSppLOCp*).

Profiles of Cloning Locality w.r.t Methods (PCLM): Kapser and Godfrey [20] provide three types of function clones based on their location – clone pairs in the same file (category 1), in the same directory (category 2) and in a different directory (category 3). They also provide the reasons, usefulness / harmfulness for each of these categories [20]. In this study we define three metrics, $PCLM(1)$ for category 1, $PCLM(2)$ for category 2 and $PCLM(3)$ for category 3 where $PCLM(i)$ gives the total number of clone pairs for category i. Furthermore, $PCLMp(i)$, the percentage clone pairs for category i is defined as follows:

$$PCLMp(i) = \frac{PCLM(i) * 100}{Total\ number\ of\ clone\ pairs\ in\ the\ system}$$

As usual similar metrics are defined with respect to lines of code (*PCLLOC and PCLLOCp*). Furthermore, the metrics values are calculated for a range of different UPI thresholds.

Profiles of Remote Cloning Locality w.r.t Methods (PRCLM): In order to study the further insights of cloning locality, we define three more metrics for the remote clone pairs (those are in different directories). Two fragments of a clone pair that are neither within the same file nor under the same directory (i.e., do not have the same parent), might have the same grandparent directory (category 1), or they might be under the same subsystem (category 2) (note that *category 1* is essentially a subset of *category 2*) or in the worst case, they might be under different subsystems (category 3). Similarly to the above definition, we define three metrics for the three categories, $PRCLM(1)$ for category 1, $PRCLM(2)$ for category 2 and $PRCLM(3)$ for category 3 where $PRCLM(i)$ gives the total number of clone pairs for category i. Furthermore, $PRCLMp(i)$, the percentage clone pairs for category i is defined as follows:

$$PRCLMp(i) = \frac{PRCLM(i) * 100}{Total\ number\ of\ different\ directory\ clone\ pairs\ in\ the\ system}$$

As usual similar metrics are defined with respect to lines of code (*PRCLLOC and PRCLLOCp*). Furthermore, the metrics values are calculated for a range of different UPI thresholds.

4. Experimental Results

In this section we provide the experimental results of this study starting from overall cloning level in C, Java and C# systems and then for each individual system in a variety of measures based on the metrics described in Section 3.4. While we provide here only the overall findings and statistical measures, the detailed results and the raw data for each systems for different UPI thresholds can be found in an online repository [29] as XML databases and HTML website.

4.1. Overall Cloning Level

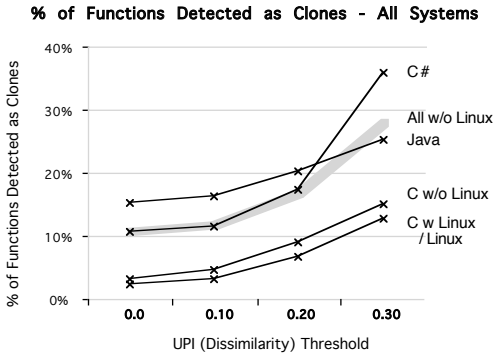
In this section we provide the overall cloning level, both in terms of number of methods and in terms of number of pretty-printed LOC (i.e., the values of the *TCM*-related metrics of subsection 3.4). Figure 2a summarizes our results for the C, Java and C# systems by the proportion of functions (or methods in the case of Java) that are cloned (i.e., *TCMp* over languages). The corresponding values for the *TCSppLOCp* metric (i.e., the proportion of clones by number of pretty-printed LOC for each language) can be found in the *%Total* rows of Table II.

The first thing we can notice is that there is significantly more function cloning in our open source Java and C# systems than in C. On average, about 15% (7.2% w.r.t LOC) of the methods in open source Java programs are exact clones - those with no changes at all (except changes in formatting, whitespace and comments), whereas only about 2.5% (1.1% w.r.t LOC) of C functions are exact clones. After detecting clones by setting a size range for the function size (see [29] for detailed results) we noticed that this is possibly in large part due to the large number of small accessor, iterator and utility methods in Java and C# programs that are not present in C.

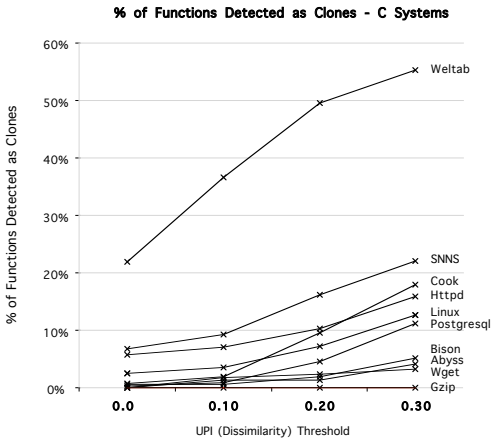
When we plot the percentage of exact clones (UPI threshold 0.0) for different clone sizes (number of pretty-printed source lines) (Figure 4), we can see that both Java and C# show similar percentages of clones for similar clone sizes, both higher than in C systems. While it is difficult to provide the exact statistics for the kinds of smaller methods for all the systems, we manually examined the small clones of the systems and found that there are many accessor methods in Java and utility / enumerator methods in C# systems. For example, the two methods (Fragments 1 and 2) in Figure 3 both appear 34 times in 34 different files in the C# system *Nant*. Similarly, in the Java system *eclipse-ant*, *Fragment 3* appears 9 times in 9 different files and *Fragment 4* appears 10 times in one *Swing* file. For C systems we did not find that many small methods, except in *httpd* and Linux.

The second thing we can notice in Figure 2a (and in the *% Total* rows of Table II for LOC) is that the effect of increasing the UPI threshold is almost identical in the languages. We can interpret this as meaning that the numbers of small changes made to cloned functions in each of these languages seems to be roughly the same in these systems. This is in some ways surprising - there is no particular reason why the pattern of changes to copied code should be similar across languages. This is even more surprising for C# systems at UPI threshold 0.3, with much more clones than C and Java systems.

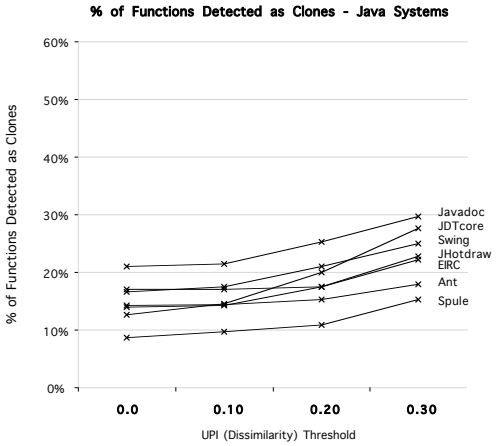
Figures 2b, 2c and 2d (also columns 3 to 6 of Table II) refine Figure 2a to show a detailed view of the same information for the individual open source C, Java and C# systems respectively.



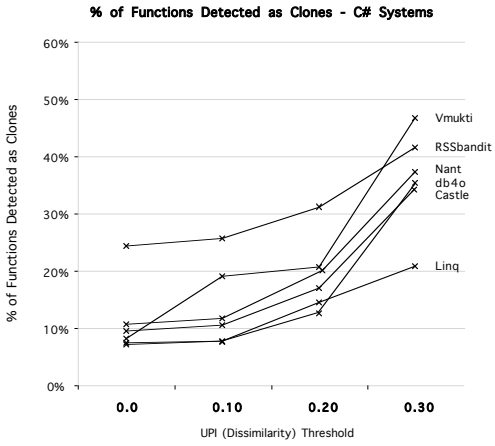
(a) Proportion of cloned functions, by language



(b) Proportion of cloned function for C systems



(c) Proportion of cloned methods for Java systems



(d) Proportion of cloned functions for C# systems

Figure 2: Proportion of cloned functions/methods in the systems

```
--------Fragment 1 (Nant)----------     -----------Fragment 2 (Nant)--------
public bool MoveNext() {                      public void Reset() {
       return _baseEnumerator.MoveNext();             baseEnumerator.Reset();
      }                                              }

---------Fragment 3 (Ant)-----------     ----------Fragment 4 (Swing)-----------
public void setValue(String value){           public boolean isEnabled() {
this.value = value; }                       return (tree != null && tree.isEnabled());}
```

Figure 3: Example of small functions/methods in C# and Java

As expected, the overall trends for each language are much like the summaries in Figure 2a (or the *% Total* rows of Table II), with lower levels of cloning in C than Java and C#. However, we can now see more. For example, we can see that several of the C systems have very little exact function cloning at all - less than 10%, and to a large extent independently of the system size (e.g., *Postgresql* and Apache *httpd* are very large, whereas *snns*, *cook* and *Weltab* are quite small). Considering the fact that Linux system is too larger than any other subject systems, we also provide the average for C language systems without Linux. It is interesting to see that there is not much difference whether we include or drop Linux in the calculations showing the fact that the cloning characteristics of the Linux Kernel is not significantly different from any other kind of C systems.

Figure 2c (also Table II for LOC) is even more interesting, because while the C and C# systems vary, we can see that the Java systems are remarkably consistent in their cloning characteristics. All begin with a relatively high level of exact method clones (between 8 and 22 percent), and in all cases allowing for changes increases the proportion only modestly. What's interesting is that this seems to be completely independent of system size, and appears to be a characteristic of the language. The only exception to this consistency seems to be *JDTcore*, which has about twice as many clones at the 0.30 dissimilarity (UPI threshold) level than exact clones. While Java and C# systems share some cloning behaviour, they are not consistent. More research will be needed to investigate this phenomenon and compare to other object-oriented languages.

In Table II we also provide the number of clone pairs and clone classes for each of the systems for varying UPI thresholds. It is interesting to note that most systems have significantly fewer clone classes than clone pairs, indicating that there are many pairs of functions in the systems that are similar, with higher numbers for Java and C# systems. It is also interesting to see that while average number of clone pairs per clone class (see the average row for each language) is more or less consistent for C and Java systems for different UPI thresholds, there is a surprising number (39.7 clone pairs per clone class) for C# systems with UPI threshold 0.3, indicating that cloning of the same functions/methods was frequent in the C# systems with a significant amount of editing/adapting in the copy/pasted code.

4.2. Clone Associated Files

The *FAWC* and *FAWCp* metrics of Section 3 address the issue of what proportion of the files in a system is associated with clones, that is, contains at least one cloned method. A system with more clones but associated with only a few files is in some sense better than a system with fewer clones scattered over many files from a software maintenance point of view. In this section, we examine the *FAWCp* metrics for each of the systems with varying UPI thresholds. Figure 5a shows the average values of *FAWCp* by language with varying UPI threshold.

We see that on average 15% of the files in the C systems, 46% of the files in the Java systems and 29% of files in the C# systems are associated with exact clones (i.e., with UPI threshold 0.0). The higher percentage of Java systems can be explained by the fact that in Java systems there are many small similar accessor methods (Figure 4). Figures 5b, 5c and 5d refine Figure 5a to show a detailed view of the same information for the individual open source C, Java and C# systems respectively. From Figure 5b we see that while most C systems have lower percentages at UPIT=0.0, a relatively small C system *Weltab* has about 51%, showing that this is a highly cloned system where 51% files are associated with at least one exact cloned method. In case of Java (Figure 5c), *Swing* shows the highest percentage, 65.9% for exact clones.

Table II. Percentage cloned LOC, clone pairs and clone classes

Lang	System	% Cloned LOC T=0.0	T=0.1	T=0.2	T=0.3	No. of Clone Pairs T=0.0	T=0.1	T=0.2	T=0.3	No. of Clone Classes T=0.0	T=0.1	T=0.2	T=0.3
C	Cook	0.3	2.0	7.7	13.3	7	18	107	280	5	12	56	98
	Httpd	2.1	4.1	6.2	9.6	183	224	322	711	107	133	195	276
	Postgresql	0.1	1.0	4.3	9.43	7	24	195	530	7	20	89	203
	Snns	3.2	6.2	13.3	18.6	109	157	343	495	63	86	143	191
	Wetlab	21.0	55.2	62.7	72.2	46	105	148	160	8	11	17	20
	Wget	0.0	1.3	1.7	2.4	0	2	4	11	0	2	2	2
	Linux	1.0	2.6	8.3	10.8	5953	7362	13813	25767	1505	2263	4613	7918
% Total C+		1.1	2.8	8.4	11.0	**#CP per Clone Class**				3.7	2.9	2.8	3.2
% Total C*		2.0	4.7	8.6	13.2	**#CP per Clone Class**				1.8	2.0	2.2	2.7
Java	Ant	5.1	5.4	6.3	9.7	363	365	374	426	92	94	101	119
	EIRC	7.2	7.2	7.7	10.9	117	117	121	149	35	35	36	47
	Javadoc	10.8	12.6	18.6	24.0	193	197	240	304	80	82	95	110
	Jdtcore	5.1	8.8	16.2	23.7	1427	1553	2126	4378	323	377	518	660
	JHotDraw	7.6	8.28	12.0	19.1	291	295	377	598	137	141	170	208
	Spule	2.0	2.7	3.1	5.9	60	64	68	113	11	13	15	19
	Swing	9.4	11.0	15	19.4	8115	8203	9978	11209	516	558	687	843
% Total Java		7.2	9.4	14.4	20.0	**#CP per Clone Class**				8.8	8.3	8.2	8.6
C#	VMukti	3.9	15.2	16.1	36.1	3	7	9	30	3	7	7	14
	Linq	3.4	3.8	8.3	12.4	427	428	523	565	4	5	16	31
	Nant	3.7	5.8	12.8	21.6	2325	2341	3519	8554	45	57	110	192
	RSS	9.8	11.7	15.3	20.6	1657	1698	2240	6405	440	469	533	605
	Castle	5.4	7.6	15.5	28.4	2110	2311	5124	21351	347	380	585	981
	db4o	4.8	5.4	10.6	26.6	1109	1149	2289	82571	391	411	652	1187
% Total C#		6.0	7.6	13.3	24.9	**#CP per Clone Class**				6.2	6.0	7.2	39.7
% Total Overall*		4.9	7.1	11.9	19.1	**#CP per Clone Class**				7.1	6.7	7.0	23.9
+ means with Linux, and * means without Linux													

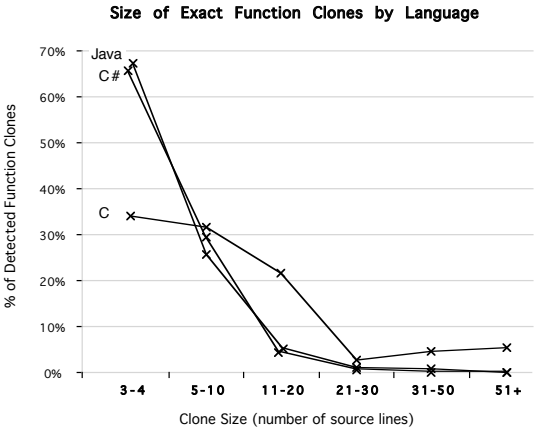
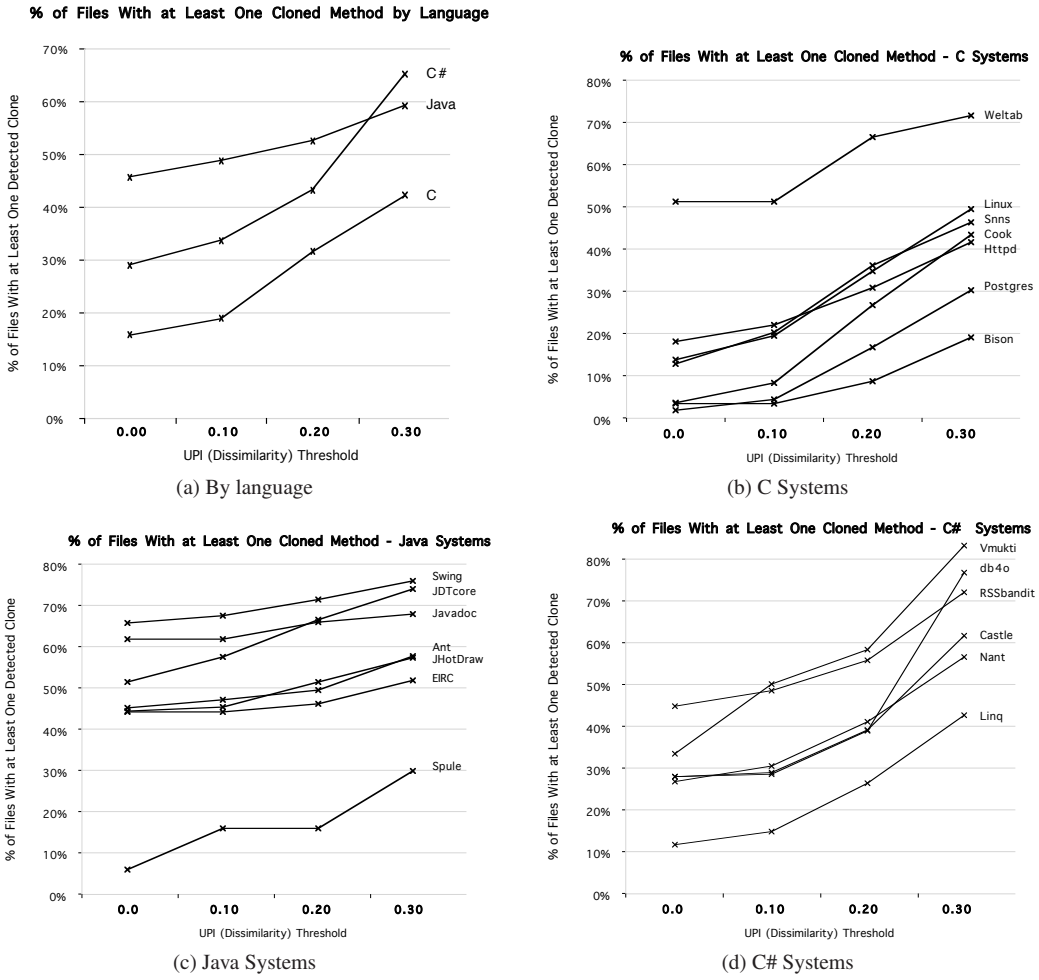


Figure 4: Percentage of exact clones by clone size



(a) By language (b) C Systems

(c) Java Systems (d) C# Systems

Figure 5: Percentage of files that are associated with cloned code

When we increase the UPI threshold to detect near-miss clones, we see that C and C# systems show a faster growing ratio than the Java systems, indicating the fact that there might be more near-miss clones in the C and C# systems than the Java systems and that the clones are scattered across different files. For example, even for the largest C system, the Linux Kernel, it increases from 14% (for UPI threshold 0.0) to 50% (for UPI threshold 0.3), whereas for the largest Java system, the *Swing*, it increases from 66% (for UPI threshold 0.0) to 76% (for UPI threshold 0.3). Most C# systems have a very high growing rate for UPI threshold 0.3 compare to other thresholds. For example, for one of the largest C# systems, the *db4o*, it only increases from 28% to 39% for UPI thresholds 0.0 to 0.2. When UPI threshold 0.3 is used, it increases to 77%, about double than with threshold 0.2.

Table III. Percentage of files that have clones over a certain percentage

Lang	System Name	UPIT	**v=0+%** Both	**v = 10+%** Meth	LOC	**v = 20+%** Meth	LOC	**v = 30+%** Meth	LOC	**v = 40+%** Meth	LOC	**v = 50+%** Meth	LOC	**v=100%** Both
C	Cook	0.0	3.8	3.5	1.4	2.8	1.0	1.7	0.7	1.4	0.7	1.4	1.4	0.7
		0.3	43.6	42.2	35.9	36.9	29.3	31.4	24.7	25.4	22.0	24.7	1.0	2.5
	Httpd	0.0	18.3	16.7	10.9	12.5	6.7	8.3	4.2	6.5	2.6	4.8	4.8	1.0
		0.3	41.7	39.3	31.9	33.5	23.8	23.6	18.1	19.6	14.9	15.5	15.5	3.8
	Postgresql	0.0	1.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		0.3	30.3	21.3	18.2	12.4	10.8	8.3	8.9	5.7	6.0	3.2	4.5	0.3
	Snns	0.0	13.0	8.7	5.8	8.7	5.1	4.3	3.6	3.6	2.9	2.9	2.2	0.0
		0.3	46.4	39.1	29.0	29.0	21.7	22.5	18.1	20.3	10.1	15.9	8.0	1.4
	Weltab	0.0	51.3	51.3	41.0	51.3	23.1	48.7	20.5	43.6	20.5	43.6	18.0	0.0
		0.3	71.8	69.2	71.8	69.2	66.7	69.2	64.1	64.1	56.4	64.1	56.4	56.4
	Avg. w/o	0.0	15.3	14.1	9.9	12.5	6.0	10.5	4.8	9.2	4.5	8.8	4.4	0.3
	Linux	0.3	42.2	38.4	34.4	32.8	28.0	28.5	25.0	25.2	20.9	22.9	16.6	12.5
	Linux	0.0	14.0	9.6	6.0	7.0	3.8	5.0	2.9	3.8	2.4	3.3	3.3	0.9
		0.3	49.6	38.1	31.0	26.0	21.5	18.0	15.7	13.3	12.1	10.7	9.5	3.1
Java	Ant	0.0	45.3	37.3	17.4	26.1	5.5	14.3	3.1	8.1	2.5	4.4	3.0	0.6
		0.3	57.8	50.9	31.1	37.9	17.4	20.5	10.6	14.3	8.1	9.9	6.8	2.5
	Javadoc	0.0	61.9	52.6	41.2	43.3	30.9	36.1	21.7	23.7	13.4	18.6	10.3	3.1
		0.3	68.0	61.9	56.7	54.6	45.4	48.5	38.1	40.2	30.9	37.1	26.8	13.4
	Jdtcore	0.0	51.5	44.2	30.4	32.8	23.0	24.2	16.8	17.7	11.7	14.4	9.8	1.7
		0.3	74.2	68.2	58.4	57.2	46.7	46.2	37.3	36.1	30.6	30.4	25.1	6.5
	JHotDraw	0.0	44.6	36.9	32.2	27.5	18.9	19.3	13.3	14.2	9.4	11.6	6.4	2.1
		0.3	57.5	53.6	51.1	45.1	39.5	36.9	30.9	27.5	22.7	21.9	17.6	5.6
	Spule	0.0	6.0	4.0	2.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		0.3	30.0	28.0	24.0	24.0	20.0	20.0	14.0	18.0	8.0	18.0	8.0	2.2
	Swing	0.0	65.9	47.3	32.6	32.1	23.4	24.9	17.4	19.1	15.9	17.1	13.8	2.2
		0.3	76.1	65.9	54.1	51.0	40.3	33.8	31.6	28.3	25.8	23.2	23.2	3.6
		0.0	45.7	37.8	26.2	27.4	17.4	20.4	12.2	14.5	8.9	11.3	7.0	1.7
	Avg.	0.3	59.4	54.3	44.9	44.1	33.9	34.2	26.1	26.9	20.1	22.2	16.7	5.1
C#	Vmukti	0	33.3	33.3	33.3	33.3	33.3	33.3	33.3	33.3	33.3	33.3	25	16.7
		0.3	83.3	83.3	83.3	83.3	75	75	66.7	66.7	58.3	66.7	41.7	25
	Linq	0	11.5	6.6	4.9	4.9	1.6	1.6	1.6	1.6	1.6	1.6	1.6	0
		0.3	42.6	37.7	32.8	31.1	23	19.7	19.7	16.4	13.1	11.5	6.6	0
	Nant	0	26.6	24.6	16.4	19.6	13.2	16.1	6.4	6.1	3.5	5	2.6	1.8
		0.3	56.4	54.7	40.9	43.9	33	40.4	29.5	33.9	28.4	29.8	26.9	8.8
	Castle	0	27.9	25.6	21.4	22.6	17.1	19.2	14.4	15.7	12.4	14.9	11.4	7.1
		0.3	61.6	59.7	54.6	55.4	48.4	50.2	43.3	43.8	37.5	41.6	34.5	23.6
	db4o	0	27.8	23.5	18.2	16.5	12.1	11.1	8.2	7.9	6	7.1	5.4	2.4
		0.3	76.6	74.5	68	66.6	55.5	57.2	46.4	49.6	38.2	45.2	32.8	17
	Avg.	0.0	28.6	25.1	20.1	21.6	16.7	18.2	14.0	14.3	12.3	13.5	10.1	5.7
		0.3	65.4	62.8	55.3	56.1	46.3	48.4	40.2	41.8	34.6	38.5	28.5	14.7

4.3. Profiles of Cloning Density

While the subsection above provides an overall view of cloning over the files in a system, one cannot immediately see which files are highly cloned or which files contain the majority of clones. In this section we provide the values for the *CRFM* and *QFCM* related metrics. Table III provides the data for the C, Java and C# systems. The first column shows the language, second and third columns give the subject systems and different UPI thresholds, while the remaining columns show the corresponding

QFCMp(v) (indicated with column *Meth*) and *QFCLOCp(v)* (indicated with column *LOC*) values. The *Avg.* row of the table shows the average values of the metrics for each of the languages of the systems. When *v=0+%* or *v=100%* (the fourth and last columns) both metrics values are same.

From the average rows of the table with *v=10+%* we can see that on average 14.1% (9.9% LOC) of the files for C systems have 10% or more of their content as exact (UPI threshold 0.0) cloned methods. For Java systems this is even higher, at 37.8% (26.2% LOC). C# systems, on the other hand, are in the middle, at 25.1% (20.1% LOC). When we increase the UPI threshold to 0.3, the C# systems become the winners, at 62.8% (55.3% LOC), with Java systems second at 54.3% (44.9% LOC) compared to 38.1% (31.0% LOC) for the C systems both in terms of methods and LOC. For higher values of v, say 50+% with UPI threshold 0.3, C# systems are still the winners, at 38.5% (28.5% LOC). However, both C and Java systems tend to have about the same percentage, at 22.9% (16.6% LOC) for C systems and 22.2% (16.7% LOC) for Java systems. One C system, *Weltab*, even has 64.1% (56.4% LOC) when *v=50+%* and 56.4% (still 56.4% LOC) when *v=100%*, once again indicating its high density of cloned code across different files. Similarly, when we look at the last column of the table at *v=100%* with UPI threshold 0.3, we see that C# systems are still the winners, at 14.7% (14.7% LOC), while C systems are second at 12.5% (12.5% LOC) and Java systems third at 5.1% (5.1% LOC). It is interesting to note that although both Java and C# are object-oriented languages and share many of the earlier cloning characteristics, they are different in this case. It is also a little surprising that 14.7% of the C# systems' files are 100% cloned code (with UPI threshold 0.3), hinting at the need for further research on this issue and comparison with other object-oriented and procedural languages. This table shows only the overall percentage for the systems. Detailed results for each file of each system can be found in our online repository [29] as an XML database.

We have also looked at the copy/paste patterns of the detected clones. Figure 6 shows examples of copy/paste changes from *Weltab*, *snns* and *Postgresql*. Assuming that cloned methods in high density cloned files have been intentionally copy/pasted, we have also compared copy/paste patterns between the high density files of the systems.

From the above discussion it is obvious that one of the interesting systems is *Weltab*. Using the NICAD interface we noticed that although there is no file in *Weltab* that is exactly similar to another, 14 files of its 39 are clones of each other even with UPI threshold 0.05, that is, they are the same but for very minor edits. As an example, in the three files *vedt.c*, *vfix.c* and *xfix.c* (each of which has two functions including one large *main* function) there are only minor differences in the *main* function, and no differences at all in the other (*acknowledge*) function. Most of the changes are in *fprintf* statements and the parameters of the *acknowledge* function when it is called, and there are also lines added/deleted in the *main* function. For other high density files we have noticed similar changes, including changes in *if-statements*, the names of functions, and so on.

Similarly, in the files *canv.c canv1.c* and *canv1a.c*, the same method *void canvw (lines, ipage, ppage, index, iward, iprec, isplt, lavcb, date, time)* is reused but the files differ slightly (e.g., changes in *fprintf* statements and some other additions/deletions of lines) in their main methods. *canv1.c* and *canv1a.c* have been detected as similar even with UPI threshold 0.05. When UPI threshold 0.20 is used all the functions of the three files are detected as clones of each other. Similarly, files *rsum.c*, *rsumxx.c*, *r51tmp.c*, *r26tmp.c*, *r11tmp.c*, *r101tmp.c* and *r01tmp.c* are not same but highly similar to each other (detected with UPI threshold 0.05), and the files *lans.c* and *lansxx.c* are highly similar.

By contrast, although *postgresql* is a substantial medium-sized system, there are very few clones. There is however, one file *scansup.c* that is 100% cloned with UPI threshold 0.20. Interestingly, it has

```
---------------Weltab ---------------
<   fprintf (stderr, "*** Skipping VFIX. Run canceled.\n");
---
>   fprintf (stderr, "*** Skipping XFIX. Run canceled.\n");
---------------Weltab ---------------
<   askchange ("", &change, &quit, FALSE, FALSE);
---
>   askchange ("", &change, &quit, FALSE, TRUE);
---------------Weltab ---------------          ---------------Weltab ----------
<   lines = LANDSCAPE;                          <  if (lines+2 > LANDSCAPE)
---                                             ---
>   lines = 60;                                 >  if (lines+2 > 60)
----------------snns -----------------          -------------Postgresql --------
<   yy_is_jam = (yy_current_state == 144);      < char *scanstr (char *s){
---                                             ---
>   yy_is_jam = (yy_current_state == 26);       > char *GUC_scanstr (char *s){
-------------------------------------           --------------------------------
```

Figure 6: Some copy/paste change examples

only one method (*backend/parser/scansup.c (original lines 30-92)*) that forms clone pair with another method in another file (*backend/utils/misc/guc-file.c (original lines 1369-1427)*). The differences are in the name of the functions, in *if statements* and couple of additions and deletions. Even with UPI threshold 0.3, no more files in this system found to be 100% cloned.

For the case of *snns*, two files, *tools/sources/lex.yyz.c* (17 methods) and *tools/sources/lex.yyy.c* (17 methods) are found to be cloned to each other with UPI threshold 0.15. Interestingly, when we use Linux *diff* they do not seem to be similar at all. But each of the extracted pretty-printed methods of the two files is either exactly similar or very similar. In fact, there are only two methods that differ between them. One pair is found cloned with UPI threshold 0.10 and the other with 0.15. Once again, the differences are minor and mostly in the *printf* statements. There is also another file, *kernel/sources/kr_pat_scan.c* that is highly similar (77.27% w.r.t methods and 92.22% w.r.t LOC) to the first two. Again, the differences are in the *if statements* and in *assignment statements*. Although in *Bison*, there are not much clones, 6 files are found completely cloned with UPI threshold 0.3. However, most files are smaller files with only one method. For the *cook* system, two files *cook/src/c_incl/os.c* and *cook/src/find_libs/os.c* (each 50LOC) are found exactly similar (detected with UPI threshold 0.0). However, only one commented line is found to be different between the files. There are 81 files in the Linux kernel that have 100% of their contents cloned either in the same files or with other files with UPI threshold 0.3. Similarly, for the other C systems we have examined the highly similar files and found the similar types of copy/ paste change patterns. Java and C# systems show similar behaviour but the methods are most cases smaller in size.

4.4. Profiles of Cloning Localization

In this section we provide the $PCLMp$ and $PRCLMp$ related metrics. The location of a clone pair is a factor in software maintenance [20, 41]. A code fragment can form a clone pair with another fragment within the same file, or it can form a clone pair with another fragment of a different file located in the same directory or with a code fragment that is located in a different file in a different directory. Kapser

```
---------Appears 2 times in /src/common/Messages.java---------------
            public void write (OutputStream out)
                  throws IOException
                  {
                      Write_int (out, Template_ID);
                      Write_int (out, Template_FatherID);
                      Write_int (out, Template_LeftBrotherID);
                      Write_int (out, Template_Type);
                      Write_String (out, Template_MetaText);
                      Write_int (out, Template_Person_ID);
                      Write_String (out, Answer_MetaAnswer);
                   }

---------Appears 7 times in /src/common/Messages.java---------------
             public void write (OutputStream out)
            throws IOException
                    {
               Write_int (out, elem.length);
               for (int i = 0; i < elem.length; ++i)
               elem[i].write (out);
             }
```

Figure 7: Two exact clones in Java system in Spule

and Godfrey [20, 19] provide a categorization of function clones based on such location differences and analyze the causes, usefulness, harmfulness and possible solutions for each kind of cloning. For example, clone pairs that appear in the same file may not be harmful as they are not physically apart and might be easily maintainable. On the other hand, clone pairs that appear in different directories, might be harmful to software maintenance as the similar fragments are hard to find and thus might not easily maintainable. They also provide cloning statistics of such clone types on the Linux Kernel *file-system subsystem version 2.4.19* and *Postgresql 7.4.2*. We further extend the similar study with more than 20 C, Java and C# systems including entire Linux Kernel with our new hybrid clone detection tool.

In Table IV we provide the percentage clone pairs for each of the different categories for the C, Java and C# systems. For each of the systems the first row represents *PCLMp* metric (i.e., w.r.t no. of methods) and the second row represents *PCLLOCp* metric (i.e., w.r.t LOC). For each of the metrics four different values are shown corresponding to UPI thresholds ranging from 0.0 (exact clone) to 0.30 (relaxed near-miss clone).

From the table we see that while there are no exact clones (UPI threshold 0.0) within the same file for C systems (except in the Linux Kernel), there are on average 18.7% (17.6% LOC) exact clone pairs within the same files for Java systems and 19.3% (18.6% LOC) for C# systems. This is particularly surprising in the Java system, *Spule*. Out of 60 exact clone pairs, 96.7% (97.2% LOC) of clone pairs occur within the same files, more specifically in file *spule/src/common/Messages.java*. Cloned methods occur between the static classes in the files. For example, two *write* methods (Figure 7) of 11 LOC and 7 LOC appear two times (original lines 1659-1669 and 1612-1622) and seven times (original lines 1796-1802, 1687-1693, 1533-1539, 1432-1438, 1391-1397, 1349-1355, and 1268-1274) respectively in file *Messages.java*. Similarly, exact clone pairs appear within the same file in other Java systems between original and abstract classes (e.g., *ant/BuildEvent.java* lines 163-165

Table IV. Percentage localization of clone pairs

Lang	System Name		Same File and Same Dir				Same Dir but Different Files				Different Dirs			
		UPIT	0.0	0.10	0.20	0.30	0.0	0.10	0.20	0.30	0.0	0.10	0.20	0.30
C	Cook	Meth	0.0	16.7	40.2	29.3	42.9	61.1	42.1	57.8	57.1	22.2	17.7	12.8
		LOC	0.0	32.3	51.3	41.1	27.3	59.8	35.9	49.1	72.7	7.8	12.7	9.8
	Httpd	Meth	0.0	2.7	15.2	33.5	6.6	7.1	9.3	7.3	93.4	90.2	75.4	59.2
		LOC	0.0	7.3	18.6	34.6	6.0	6.9	10.7	7.9	94.0	85.8	70.7	57.4
	Postgresql	Meth	0.0	62.5	89.7	87.6	14.3	4.2	2.5	6.9	85.7	33.3	7.6	5.5
		LOC	0.0	81.3	89.5	86.1	5.3	0.4	4.7	8.7	94.7	18.3	5.7	5.2
	Snns	Meth	0.0	15.9	45.2	54.3	62.4	55.4	37.9	33.3	37.6	28.6	16.9	12.3
		LOC	0.0	33.8	50.4	59.8	48.7	38.3	32.7	27.6	51.3	27.9	16.8	12.5
	Weltab	Meth	0.0	0.0	2.7	3.1	100.0	100.0	97.3	96.9	0.0	0.0	0.0	0.0
		LOC	0.0	0.0	0.4	0.5	100.0	100.0	99.5	99.5	0.0	0.0	0.0	0.0
	Avg. w/o Linux	Meth	0.0	24.6	44.7	49.8	37.7	46.3	35.7	35.2	45.6	29.1	19.6	15.0
		LOC	0.0	35.2	48.3	52.5	31.2	41.5	34.0	33.4	52.1	23.3	17.7	14.2
	Linux	Meth	0.1	2.7	12.2	22.5	55.5	53.3	41.2	33.5	44.4	44.1	46.6	44.0
		LOC	0.3	8.6	20.2	28.0	58.9	52.5	40.0	34.7	40.8	39.0	39.8	37.2
Java	Ant	Meth	4.4	4.7	5.1	6.6	73.0	72.6	72.5	69.3	22.6	22.7	22.5	24.2
		LOC	4.2	5.2	6.1	11.6	75.2	73.3	72.3	62.6	20.6	21.5	21.6	25.8
	javadoc	Meth	19.7	20.8	19.6	22.7	72.0	71.1	69.2	62.8	8.3	8.1	11.3	15.5
		LOC	14.3	22.5	25.8	25.8	76.3	69.4	63.4	60.2	9.4	8.0	10.8	14.1
	Jdtcore	Meth	0.9	3.1	11.8	41.9	75.8	72.9	68.9	46.4	23.3	24.0	19.3	11.7
		LOC	0.6	13.0	34.3	59.4	73.9	56.5	42.0	28.2	22.5	30.5	21.6	12.4
	JHotDraw	Meth	5.8	5.8	6.6	13.7	51.6	52.2	47.5	39.1	42.6	42.0	45.9	47.2
		LOC	4.5	4.2	7.5	18.7	50.5	53.3	46.0	36.6	45.0	42.5	46.6	44.6
	Spule	Meth	96.7	90.6	91.2	77.0	3.3	4.0	8.8	23.0	0.0	0.0	0.0	0.0
		LOC	97.2	86.1	87.3	64.6	2.9	13.9	12.7	35.4	0.0	0.0	0.0	0.0
	Swing	Meth	3.5	3.7	5.3	9.6	87.5	87.2	83.8	77.8	9.0	9.1	10.9	12.6
		LOC	2.4	3.2	8.3	18.5	90.0	88.5	81.0	68.7	7.6	8.3	10.7	12.9
	Avg.	Meth	18.7	18.4	20.1	25.3	61.2	60.7	59.4	54.5	20.1	20.1	20.5	20.4
		LOC	17.6	19.2	24.5	29.7	62.4	60.5	55.1	50.4	19.5	20.4	20.1	20.0
C#	Vmukti	Meth	0.0	0.0	0.0	50.0	0	0.0	0.0	13.3	100.0	100.0	100.0	36.7
		LOC	0.0	0.0	0.0	45.2	0	0.0	0.0	17.7	100.0	100.0	100.0	37.1
	Linq	Meth	98.4	98.1	95.0	89.9	1.6	1.9	5.0	10.1	0.0	0.0	0.0	0.0
		LOC	99.0	98.6	94.2	89.0	0.9	1.4	5.8	11.1	0.0	0.0	0.0	0.0
	Nant	Meth	0.0	0.3	2.3	3.0	12.7	12.9	12.8	11.1	87.1	86.7	84.9	85.8
		LOC	0.0	3.7	11.0	12.6	14	14.3	14.1	12.1	86.0	82.0	74.9	75.4
	Castle	Meth	1.9	9.4	48.2	23.4	56.5	51.9	31.9	13.7	42.3	38.7	19.9	62.9
		LOC	1.1	18.5	59.3	35.2	58.4	48.2	27.4	14.5	40.5	33.2	14.4	50.3
	db4o	Meth	2.3	3.1	4.6	3.5	44.5	44.5	39.4	7.2	53.2	52.4	56.0	89.3
		LOC	1.9	4.8	5.5	4.8	46.6	45.7	39.9	8.0	51.6	49.6	54.6	87.1
	Avg.	Meth	19.3	20.8	28.3	33.0	24.0	23.3	19.5	14.5	56.7	55.9	52.3	52.4
		LOC	18.6	22.8	31.8	36.2	24.4	22.6	18.5	15.3	57.0	54.7	49.9	48.5

and 93-95), as inlined functions (e.g., *jdtcore/src/internal/compiler/Compiler.java* lines 148-151 and 84-87) and so on. In C# system, *Linq*, there are also many exact clone pairs within the same file. For example, two functions *SendPropertyChanged* and *SendPropertyChanging* (Figure 8) appear 21 times each in file *Linq/Source1712/Source/Entities/DB/DB.designer.cs* as protected virtual functions in original lines (390-396, 4038-4044, 3976-3982, 3693-3699, 3062-3068, 2810-2816, 2696-2702, 2582-2588, 2468-2474, 2253-2259, 2085-2091, 1892-1898, 1713-1719, 1599-1605, 1497-1503, 1329-1335, 1161-1167, 1015-1021, 889-895, 721-727, 570-576) and (4030-4036, 3968-3974, 382-388, 3685-

```
------Appears 21 times in Source/Entities/DB/DB.designer.cs--------
protected virtual void SendPropertyChanged(String pName)
  {
    if ((this.PropertyChanged != null))
      {
     this.PropertyChanged(this, new PropertyChangedEventArgs(pName));
       }
   }

------Appears 21 times in Source/Entities/DB/DB.designer.cs--------
protected virtual void SendPropertyChanging()
   {
      if ((this.PropertyChanging != null))
        {
           this.PropertyChanging(this, emptyChangingEventArgs);
           }
    }
```

Figure 8: Two exact clones in the C# system Linq

3691, 3685-3691, 2802-2808", 2688-2694, 2574-2580, 2460-2466, 2245-2251, 2077-2083, 1884-1890, 1705-1711, 1591-1597, 1489-1495, 1321-1327, 1153-1159, 1007-1013, 881-887, 713-719, 562-568) respectively. While most of the clone pairs are smaller in size, exact cloning within the same file might be interesting both from software maintenance and language design points of view.

As an exception to the C systems, the Linux Kernel has one file (*/drivers/net/fec.c*) that has six exact clone pairs in it. In fact there are two exact clone classes in this file. In the first class, function fec_get_mac (32 LOC) appears three times (lines 1744-1775, 1595-1626 and 1464-1495) and in the second class, function fec_set_mii (16 LOC) also appears three times (lines 1727-1742, 1577-1593 and 1446-1462) as static inline functions.

However, when we detect near-miss clones by allowing a higher UPI threshold, we see that the metrics values grow at a higher rate for the C systems than the Java and C# systems. For example, when UPI threshold is 0.3, on average 49.8% (52.5% LOC) of clone pairs of the C systems occur within the same file compared to only 25.3% (29.7% LOC) of the clone pairs in the Java systems. C# systems show 33.0% (36.2%), a little bit higher percentages than Java systems. If we have a close look at the individual systems we also see higher ratios for most of the C systems than the Java and C# systems. In the case of the C system *Wget* it actually reaches to 90.9% (92.7% LOC). However, *Wget* is a small system, has only 11 clone pairs in 2 classes with UPI threshold 0.3, and only 3 of its 21 files are associated with clones. Taking a closer look, we found that of the 2 clone classes, one contains 5 cloned methods (original lines 503-539, 463-499, 389-425, 350-385, and 428-460) and all are from the same file *wget/src/ftp-basic.c*.

However, a reasonably large system, *Postgresql* (530 clone pairs in 203 classes with UPI threshold 0.3), also shows a higher percentage (87.6% methods and 86.1% LOC) of clone pairs within the same files with UPI threshold 0.3. As with *Wget*, it has also higher frequency of clone classes in the same file. For example, in file *postgresql/src/backend/utils/adt/float.c*, there are six similar methods (original lines 952-965, 935-948, 900-913, 865-878, 848-861, and 831-844) of 14 LOC differing only in their function names and built-in function calls (e.g., *tan* changes to *sin*, *cos* to *acos*, and so on).

Both C and Java systems tend to have a higher percentage of exact clones (UPI threshold 0.0) within the same directory but in different files than the C# systems. Even the largest C system, the Linux Kernel, has 55.5% (58.9% LOC) of its exact clone pairs in the same directory (but different files). The largest Java system, the Java 2 SDK *Swing*, has even more, 87.5% (90.0% LOC). Among the C# systems, the *Castle*, one of the biggest C# systems in this study, shows the highest, 56.5% (58.4% LOC). C# systems are the winners for exact clone pairs of different directories though, 62.9% (63.2% LOC), compare to 45.5% (50.5% LOC) for C systems and 20.1% (19.5% LOC) for Java systems.

When we look for near-miss clones by increasing the UPI threshold, we see interesting phenomenon for different language paradigms. While these percentages of different directory clone pairs tend decrease significantly for procedural C systems, from 45.6% (52.1% LOC) to 15.0% (14.2% LOC), they tend to remain constant or even increase for the object-oriented Java and C# systems. This phenomenon for the C systems indicates that clone pairs either form within the same file or between different files of the same directories. After a close look on the same file and same directory percentages, we see that clone pairs for the C systems actually tend to form within the same files with increasing UPI thresholds, from 0.0% (0.0% LOC) to 49.8% (52.5% LOC), confirming the fact that there are significant amount of near-miss clone pairs within the same files in the C systems.

In order to study the further insights of cloning locality, we also provide $PRCLMp$ related metrics, i.e., the locality of the remote clone pairs (i.e., those are in different directories). Two fragments of a different directory clone pair (i.e., the fragments have different parent directories) might share the same grandparent directory or they might be under the same subsystem or in the worst case, they might be from different subsystems. As discussed earlier, the more closer the fragments of clone pairs are, the more easily maintainable they might be. Table V shows the data for the C, Java and C# systems of this study that have different directory clone pairs.

From the *Avg.* rows of the table, we see that exact clone pairs (UPI threshold 0.0) those are not within the same file or in different files under the same directory, tend to have same grandparents or subsystems. For example, 36.6% (38.5% LOC) of different directory exact clone pairs of C# systems have the same grandparents, while only 14.7% (14.3% LOC) are under different subsystems. C and Java systems tend to have similar percentages to C#. One should also note that clone pairs those have the same grandparents, essentially are under the same subsystems.

When we increase the UPI threshold for detecting near-miss clones, we see that both for C and C# systems, the percentages of same grandparent clone pairs tend to remain constant or even increase while Java systems tend to decrease (from 37% to 31%). Percentages of clone pairs for the other two metrics (same subsystem and different subsystems) also tend to remain constant for C and C# systems and with increasing ratios for Java systems. Figures 9a and 9b show an overview of the localization of clone pairs for UPI threshold 0.3.

Since we consider all functions with 3 LOC or more in pretty-printed format, one might argue that the findings are biased on the size of the functions. However, it does not seem so when we look the values for LOC. In almost all cases, both the method and LOC metrics values are very close, showing that a significant proportion of files in each system have a significant proportion of similar code in the files themselves. The commonalities and variabilities of cloning properties of C# with the C and Java systems might be exacerbated by the fact that C# a multi-paradigm programming language that encompasses functional, imperative, generic, object-oriented (class-based), and component-oriented programming disciplines. When we look the source of the C# systems, we also see the usage of different programming paradigms.

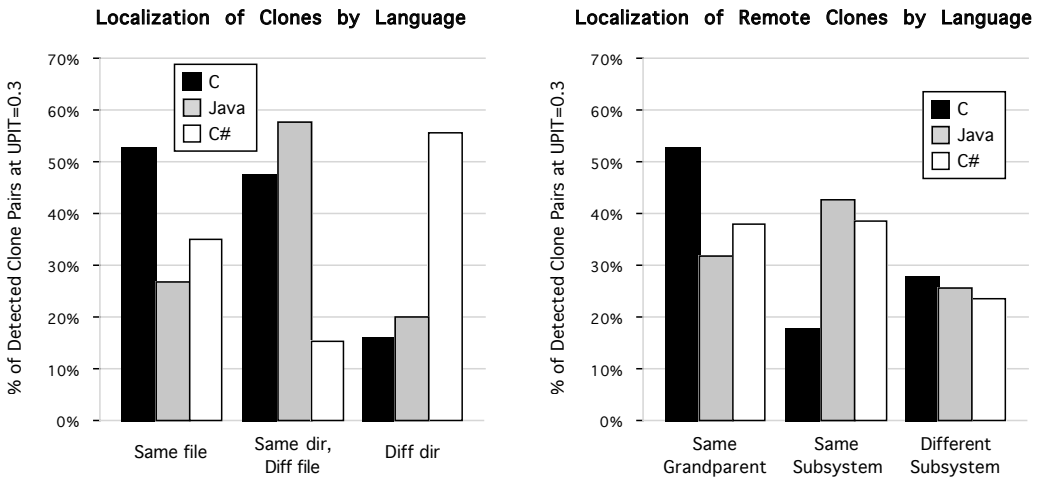
Table V. Percentage localization of remote clone pairs

Language	System	With Respect To	UPIT=0.0			UPIT=0.1			UPIT=0.2			UPIT=0.3		
			Same Grandparent	Same Subsystem	Different Subsystem	Same Grandparent	Same Subsystem	Different Subsystem	Same Grandparent	Same Subsystem	Different Subsystem	Same Grandparent	Same Subsystem	Different Subsystem
C	Cook	Meth	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0
		LOC	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0
	Httpd	Meth	76.6	22.8	0.6	74.3	22.8	3.0	73.7	21.4	4.9	63.4	23.5	13.1
		LOC	58.6	41.2	0.2	51.8	42.1	6.1	57.7	35.3	7.0	57.6	31.8	10.6
	Postgresql	Meth	0.0	100.0	0.0	25.0	75.0	0.0	40.0	60.0	0.0	51.7	48.3	0.0
		LOC	0.0	100.0	0.0	57.3	42.7	0.0	45.9	54.1	0.0	73.4	26.6	0.0
	snns	Meth	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100
		LOC	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100
	Avg. w/o Linux	Method	44.2	30.7	25.1	49.8	24.4	25.7	53.4	20.3	26.2	53.8	17.9	28.3
		LOC	39.6	35.3	25.0	52.3	21.2	26.5	50.9	22.4	26.7	57.7	14.6	27.6
	Linux	Meth	31.1	64.2	4.7	31.1	64.4	4.5	36.9	59.2	3.9	35.7	59.4	4.9
		LOC	30.7	65.7	3.5	33.2	62.8	4.0	41.0	55.0	4.0	40.9	54.5	4.6
Java	Ant	Meth	47.6	45.1	7.3	47.0	45.8	7.2	47.6	45.2	7.1	44.7	47.6	7.8
		LOC	48.1	44.8	7.1	45.1	48.3	6.6	49.9	44.1	6.1	47.4	47.4	5.1
	javadoc	Meth	68.8	0.0	31.3	68.8	0.0	31.3	63.0	0.0	37.0	43.2	0.0	56.8
		LOC	58.9	0.0	41.1	58.9	0.0	41.1	51.2	0.0	48.8	33.9	0.0	66.1
	Jdtcore	Meth	20.1	71.5	8.4	20.7	71.5	7.8	20.0	71.8	8.3	20.8	71.6	7.6
		LOC	16.8	74.5	8.7	16.9	77.9	5.1	13.9	81.6	4.6	14.8	80.8	4.4
	JhotDraw	Meth	74.2	25.8	0.0	74.2	25.8	0.0	77.5	22.5	0.0	67.7	32.3	0.0
		LOC	75.9	24.1	0.0	75.9	24.1	0.0	81.4	18.6	0.0	72.6	27.4	0.0
	Swing	Meth	13.3	12.5	74.2	13.2	13.3	73.6	9.8	10.0	80.2	9.0	8.3	82.7
		LOC	9.3	14.8	75.8	8.8	19.0	72.2	7.7	12.5	79.8	6.4	9.0	84.6
	Avg.	Meth	38.1	41.7	20.2	38.1	41.9	20.0	37.1	40.8	22.1	31.6	42.6	25.8
		LOC	35.5	42.4	22.1	34.9	44.3	20.8	34.6	42.2	23.2	29.7	43.6	26.7
C#	Vmukti	Meth	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0
		LOC	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0
	Nant	Meth	5.3	94.5	0.2	5.4	94.4	0.2	5.6	94.2	0.2	5.8	93.9	0.3
		LOC	5.6	94.2	0.2	6.4	93.4	0.2	6.2	93.6	0.2	6.1	93.7	0.2
	Castle	Meth	12.1	67.4	20.5	12.1	67.5	20.4	11.3	66.8	21.9	4.9	23.1	71.9
		LOC	16.3	65.8	17.9	16.0	66.5	17.5	14.1	65.9	19.9	5.2	23.9	71.0
	db4o	Meth	34.2	20.8	44.9	34.4	20.4	45.2	46.2	25.1	28.7	39.6	29.5	30.9
		LOC	36.6	20.6	42.8	37.9	19.1	43.0	48.9	27.0	24.1	39.2	29.9	30.9
	Avg.	Meth	36.6	48.7	14.7	36.5	48.7	14.8	40.0	48.3	11.7	38.1	38.8	23.1
		LOC	38.5	47.2	14.3	38.4	47.2	14.4	40.8	48.0	11.3	37.3	39.2	23.5

5. Related Work

Empirical study of clones in open source systems is not a new topic. When a new clone detection technique is published, it normally comes with an empirical study (at least in part). However, these studies focus on validating the proposed methods [34] rather than on the subject systems.

Several tool comparison studies have used open source systems for comparing different tools [34]. Of them, the Bellon et al. experiment [7, 6] is the most extensive to date, using four C and four Java



(a) % localization of clone pairs, by language

(b) % localization of remote clone pairs, by language

Figure 9: Percentage localization of clone pairs at UPIT = 0.3

systems to compare several state-of-the-art tools. Although we have used the subject systems from Bellon's experiment as a part of our study, our study differs in using a new hybrid clone detection tool, in the size of the subject systems analyzed (e.g., the entire Linux Kernel) and in providing the cloning status of the subject systems themselves in several different dimensions.

Kapser and Godfrey have conducted extensive empirical studies with Apache *httpd*, the Linux file system and several other open source systems. They provide a detailed categorization of code clones in the form of a taxonomy [20], propose a new analysis framework [19] and give an in-depth study on the harmfulness / usefulness of cloning [18]. Our study differs in that we focus on the comprehensive cloning status of a wide variety of different systems in different languages, whereas they focus on the maintenance implications of cloning.

Empirical studies of cloning in the Linux Kernel have also been carried out by several other researchers. Of them, Casazza et al. [9] and Antoniol et al. [4] provide interesting findings, but they focus on clone evolution, whereas we focus on the occurrence of copy/paste clones. Kim et al. [22] also studied the evolution of code clones in several systems and concluded that programmers often intentionally practice code cloning. Similarly Lozano [25] studied the evolution of clones to assess the impact of source code flaws in changeability. Jiang and Hassan [13] also used the Linux Kernel as an example for their framework for understanding cloning in very large systems. Uchida et al. [40] studied code clones in 125 open source C packages for software analysis, and Jia et al. [14] examined the effectiveness of a novel method that extends lexical analysis using the notion of local dependence in several open source projects.

Al-Ekram et al. [3] have also conducted a promising empirical study on cloning, focussing on C/C++ systems from two different domains. They examined different clone types (e.g., accidental clones) by analyzing clones across systems in the same domain, whereas we have studied a wide variety of systems and concentrated on copy/paste function clones of individual system. Krinke [23] has conducted an

empirical study with five C/C++/Java systems, focussing on consistent and inconsistent changes to exact code clones in different versions of the subject systems. The most closely related work to ours is the work of Rajapakse and Jarzabek [27] which was also one of the motivations of our study. However, they studied cloning in a different domain, web applications, and have looked at only exact clones.

6. Conclusion

In this paper we have provided an empirical study of function clones in several C, Java and C# open source software systems of varying size, including Apache *httpd* and the entire Linux Kernel, using the new hybrid clone detection method NICAD. We have provided cloning statistics for these systems in several dimensions and made the detailed results available in an online repository [29]. These results can potentially be used as a benchmark for evaluating other clone detection tools. Due to limited space, we have omitted the results for some small systems and for difference thresholds 0.1 and 0.2. Detailed statistics for all systems and for all thresholds can be found elsewhere [29, 35].

As regards the research questions raised in the introduction, for the first question, we can say that our results indicate a large number of exact function clones in these open source systems. We also see much higher percentages of near-miss clones, indicating significantly higher numbers of near-miss clones than exact clones in these systems. For the second question, we can say that we observed many more exact function clones in object-oriented Java and C# systems than in C systems. However, the effect of increasing the UPI threshold for near-miss clones was almost identical regardless of language paradigm. For the third question, we observed no significant differences in cloning related to the size of the systems. Even though Linux is huge, its cloning characteristics seem to be typical of other C systems. Similarly, the largest Java system *Swing* and the largest C# system *db4o* seem to be representative of the cloning characteristics of other systems written in their corresponding languages.

Threats to Validity: One of the major threats to the results of this study is the lack of a sound definition of code clones. While one can precisely define exact clones, there is no agreed upon definition of near-miss clones. In this study we have used a dissimilarity threshold on the standard pretty-printed code as a measure of near-miss clones. While this gives good results, we cannot be sure that these are definitively the right set for software maintenance activities such as refactoring. The second threat to this study is the limited number of samples. However, to our knowledge this is the first study in clone detection research that considers more than 20 open source systems of different languages.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable comments, suggestions and corrections in improving the paper. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada and by an IBM International Faculty Award.

REFERENCES

1. The Abyss: http://abyss.sourceforge.net/ (Dec 2007)
2. The Apache-httpd: http://httpd.apache.org/ (April 2008)
3. R. Al-Ekram, C. Kapser and M. Godfrey. Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. In *ISESE*, pp. 376-385, 2005.
4. G. Antoniol, U. Villano, E. Merlo and M. Di Penta. Analyzing Cloning Evolution in the Linux Kernel. *Information and Software Technology*, 44 (13):755-765, 2002.

5. L. Aversano, L. Cerulo, and M. Di Penta. How Clones are Maintained: An Empirical Study. In *CSMR*, pp. 81-90, 2007.
6. S. Bellon and R. Koschke. Detection of Software Clone: Tool Comparison Experiment: http://www.bauhaus-stuttgart.de/clones/ (December 2007).
7. S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577-591, 2007.
8. E. Burd, J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *SCAM*, pp. 36-43, 2002.
9. G. Casazza, G. Antoniol, U. Villano, E. Merlo and M. Di Penta. Identifying Clones in the Linux Kernel. In *SCAM*, pp. 90-97, 2001.
10. R. Falke, R. Koschke and P. Frenzel. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empirical Software Engineering*, Vol. 13: 601-643, 2008.
11. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
12. The Gzip-1.2.4 http://www.gzip.org/ (Feb 2008).
13. Z. Jiang and A. Hassan. A Framework for Studying Clones in Large Software Systems. In *SCAM*, pp. 203-212, 2007.
14. Y. Jia, D. Binkley, M. Harman, J. krinke and M. Matsushita. KClone: A Proposed Approach to Fast Precise Code Clone Detection. In *IWSC*, 5 pp., 2009.
15. The JHotDraw: http://www.jhotdraw.org/ (June 2006)
16. J. Johnson. Visualizing Textual Redundancy in Legacy Source. In *CASCON*, pp. 171-183, 1994.
17. E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner, Do Code Clones Matter? In *ICSE*, pp. 485-495, 2009.
18. Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering*, Vol. 13(6):645-692 (2008).
19. C. Kapser and M. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *JSME: Research and Practice*, 18(2):61-82, 2006.
20. C. Kapser and M. Godfrey. Toward a Taxonomy of Clones in Source Code: A Case Study. In *ELISA*, pp. 67-78, 2003.
21. The Linux-2.6.24.2: http://www.linux.org/ (March 2008)
22. M. Kim, V. Sazawal, D. Notkin and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE*, pp. 187-196, 2005.
23. J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *WCRE*, pp. 170-178, 2007.
24. Z. Li, S. Lu, S. Myagmar and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176-192, 2006.
25. A. Lozano. A Methodology to Assess the Impact of Source Code Flaws in Changeability, and its Application to Clones. In *ICSM Doctoral Symposium*, pp. 424-427, 2008.
26. J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *ICSM*, pp. 244-253, 1996.
27. D. C. Rajapakse and S. Jarzabek. An Investigation of Cloning in Web Applications. In *WWW*, pp. 924-925, 2005.
28. M. Rieger, S. Ducasse and M. Lanza. Insights into System–Wide Code Duplication. In *WCRE*, pp. 100-109, 2004.
29. C.K. Roy and J.R. Cordy. JSME'09 Clone Results: http://www.cs.queensu.ca/home/stl/download/NICADOutput/ (June 2009).
30. C.K. Roy and J.R. Cordy. An Empirical Study of Function Clones in Open Source Software Systems. In *WCRE*, pp. 81-90, 2008.
31. C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*, pp. 172-181, 2008.
32. C.K. Roy, J.R. Cordy and R. Koschke. Comparison and Evaluation of Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74 (2009) 470-495, 2009.
33. C.K. Roy and J.R. Cordy. Scenario-Based Comparison of Clone Detection Techniques. In *ICPC*, pp. 153-162, 2008.
34. C.K. Roy and J.R. Cordy. *A Survey on Software Clone Detection Research*. Queen's School of Computing TR 2007-541, 115 pp., 2007.
35. C.K. Roy. Detection and Analysis of Near-miss Software Clones. Ph.D. Thesis, Queen's School of Computing, 263 pp., 2009 (submitted).
36. C.K. Roy and J.R. Cordy. A Mutation / Injection-based Automatic Framework for Evaluating Clone Detection Tools. In *Mutation'09*, pp. 157-166, 2009.
37. F. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques. In *ELISA*, 12pp, 2003.
38. F. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques from a Refactoring Perspective. In *ASE*, pp. 336-339, 2004.
39. The SOURGEFORGE.NET, http://sourceforge.net/ (last access February 2009).
40. S. Uchida, A. Monden, N. Ohsugi and T. Kamiya. Software Analysis by Code Clones in Open Source Software. *Journal of Computer Information Systems*, XLV(3):1-11, 2005.
41. Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In *METRICS*, pp. 67-76, 2002.