Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations

Akito Monden and Satoshi Okahara, Nara Institute of Science and Technology, Japan
Yuki Manabe, Osaka University
Kenichi Matsumoto, Nara Institute of Science and Technology, Japan

Programmers often unwittingly violate open source software licenses by reusing code fragments, or clones. The authors explore metrics that can reveal the existence or absence of code reuse and apply these metrics to 1,225 open source product pairs.

TO INCREASE PRODUCTIVITY, programmers often reuse open source software (OSS) code as a part of their product. Although code reuse is generally sound practice, software companies are often unaware of how much reuse occurs, particularly if they outsource development. As more reusable OSS code becomes available online, it’s becoming increasingly important for companies to inspect their software products for possible OSS code violations. This is no trivial task, since there are already myriad OSS licenses. Indeed, the violation problem is so pervasive that OSS developer-side organizations such as the Software Freedom Law Center (www.softwarefreedom.org) and gpl-violations.org have begun helping developers detect OSS license violations in commercial products.

Several services are available for OSS code detection and license identification and management. Black Duck Software’s Protex analyzes source code using code print technology, comparing the source code against an OSS repository of more than 200,000 products (www.blackducksoftware.com/protex). Palamida detects reused OSS source code through multipattern searching that involves identifying matches of code fragments, or clones, and then ranking the matches according to relevance (www.palamida.com). Unfortunately, as far as we know, no one has published a performance evaluation of these services with quantifiable metrics, such as recall and precision or false-positive/negative detection rate.

We also see flaws in the core technology that these detection services use, which is to search for code clone matches between two programs and judge reuse if the programs share a large number of clones or a large enough single clone. These programs don’t account for the case in which a programmer accidentally introduces clones that don’t expose any licensing issue. Also, the question is still open as to what clone number or size translates to a violation. That is, what number of clones or what single clone size is the threshold for determining that the suspected program is guilty or not guilty of a licensing violation?

We decided to tackle this open problem by answering two questions:

- What metrics are appropriate in evaluating the number and size of code clones between two programs?
- What’s the lower bound of code
clone measurement needed to conclude that the suspected program is guilty, and what is the upper bound needed to conclude that it isn’t?

To answer the first question, we reviewed several potential clone metrics including ones from www.ccfinder.net, such as the ratio of similarity between another file and coverage. From these, we selected three clone-based measures: maximum length of clones (MLC), number of clone pairs (NCP), and clone-based local similarity (LSim), which looks at the percentage of duplication within a suspicious pair.

To answer the second question, we first established the framework in Figure 1 for defining guilty and not guilty and then used our metrics to determine the upper and lower bounds. To experimentally identify these bounds, we analyzed 1,225 pairs of OSS products for reuse-based clones.

**Clone Detection**

In the context of our work, code clones are exact or nearly exact duplicated lines of code between a pair of source programs. These *interproduct* clones appear when one of the programs has reused the other’s code or when both have reused the same piece of third-party code. We don’t distinguish between these reuse conditions because companies must identify both cases to comply with software licensing.

Accidental interproduct clones are a case that must be distinguished, however, because they don’t expose any licensing issues. Programmers can introduce accidental interproduct clones by using code snippets, for example, which are essentially mental macros—definitional computations that a programmer frequently codes in a regular style, such as payroll tax, queue insertion, or data structure access. Fortunately, accidental clones tend to be smaller than reuse-based clones, and because size is one of our metrics, we are able to separate them.

Most existing and proposed tools aim to detect three types of interproduct clones:

- **Type 1.** An exact copy with no modifications except white space and comments.
- **Type 2.** A syntactically identical copy, with only variable, type, or function identifiers changed.
- **Type 3.** A copy with further modifications, with statements changed, added, or removed.

We’re concerned only with the detection of Type 2 clones. The detection of Type 1 clones isn’t sufficient for our goal because programmers often modify reused code fragments slightly to adapt them to a new environment or purpose. Type 3 clone detection isn’t suitable because the definition of Type 3 clones is vague, and there’s no clear consensus on a suitable equivalence or similarity measure for these clones.

To detect Type 2 clones, we used CCFinderX, which is a major upgrade of the CCFinder clone detector (available at www.ccfinder.net). CCFinderX compares the token sequences of lines of source code and can detect code in systems up to a million lines within affordable computation time and memory requirements.

**Metrics Definition**

By applying MLC, NCP, and LSim, we can distinguish reuse-based clones from accidental ones. We define these metrics as

- **MLC**—number of tokens of the largest clone pair detected between two programs;
- **NCP**—sum of all clone pairs greater than 30 tokens; and
- **LSim**—percentage of duplication within the most suspicious source file pair (the one with the largest clone).

LSim is a valuable metric for detecting reuse when a programmer has copied one or more files from the other product. Given the task of inspecting programs A and B, \( LSim(A,B) \) is

\[
LSim(A,B) = \frac{2 \times MLC(A,B)}{|a| + |b|} \times 100, \tag{1}
\]
FOCUS: SOFTWARE PROTECTION

OPEN SOURCE PRODUCTS EVALUATED

We evaluated 50 products from among those listed at the Free Software Directory:

- abcm2ps-3.7.20
- acme-2.0.2
- aide-0.9
- aliens_V1.0.0
- asteroids3D
- avdtools-0.3
- barcode-0.98
- battstat_applet-2.0.11
- bc-1.06
- beecrypt-4.1.2
- bmi-1.3.1
- calcoo-1.3.16
- ccvssh-0.9.1
- cddc-0.6.6
- cdparanoia-III-alpha9.8
- cdtools-2.0
- cflow-1.2
- check-0.8.4
- chemtool-1.6.11
- cinepaint-0.19-0
- cvsgraph-1.5.2
- cvsp-1.3.3
- danpei-2.9.6
- dap-3.6, Deki_Wiki_8.08.1_Kilin_ Woods_source
dox-1.0beta4
easytag-1.1
electrocardiognosis
euler-1.60
fmd-1.392
ffproxy-1.6
filemanager-0.972

firestarter-1.0.3
fnord-1.9
FreeCAD-0.33
gaby-2.0.3
gcalculator-1.2.4
gbonds-2.0.2
gcal-3.01
geomview-1.8.1
gforge-3.0
gimp-2.3.19
glabels-2.0.3
glame-2.0.1
glom-0.9.8
gmandel-1.1.0
gnmmusic-1.1.91
gnofin-0.8.4
gnub-0.14.3
gnugo-3.6

do not contain reused code.)

Of the 1,225 pairs, 796 included one or more clones. Using the results of a

two-month inspection by an expert programmer, we identified 121 product pairs of the

original 796 as including reuse-based clones. We applied the following procedure
to identify the 121 pairs:

- For each pair, an expert programmer (one of us) picked the largest clone
  and judged on the basis of his experience if it was likely to be produced
cidentally. If no, the clone was considered reuse-based (guilty); if yes, it
  was deemed accidental (not guilty).
- If the clone was guilty, the programmer went to the next pair.
  - If the clone wasn’t guilty, the programmer picked the next largest
    clone and continued judging until clone length reached 30 tokens.

To identify the lower bound of each clone metric, for a given metric value
(which we consider a temporal as opposed to a true lower bound), we labeled
all product pairs with a greater metric value as potentially guilty. We then computed precision and recall.

Precision is the number of correctly labeled pairs divided by the total number
of pairs potentially labeled guilty. Recall is the number of correctly labeled
products divided by the total number of pairs that are actually guilty. The lower
bound is then the lowest metric value that meets 100 percent precision: no
pair classified as guilty is misclassified. Similarly, to identify each metric’s upper
bound, we labeled all pairs with a lower metric value as belonging to not guilty
and identified the largest value that meets 100 percent precision—no pair
classified as not guilty is misclassified.

where \(a\) and \(b\) comprise a file pair with

the largest clone between \(A\) and \(B\),

\(MLC(A,B)\) is the length of the largest

clones, and \(|a|\) and \(|b|\) are the lengths of

\(a\) and \(b\) as number of tokens. If a file

pair \(a\) and \(b\) are identical, then \(LSim\)

becomes 100 percent.

We hypothesized that the larger the value of \(MLC\), the greater the possibility
of code reuse and that the greater the value of \(NCP\), the greater the possibility
of code reuse.

Experiment Setting

To test our metrics, we selected 50 OSS products from among products divided by the total number of
pairs potentially labeled guilty. Recall is the number of correctly labeled
products divided by the total number of pairs that are actually guilty. The lower
bound is then the lowest metric value that meets 100 percent precision: no
pair classified as guilty is misclassified. Similarly, to identify each metric’s upper
bound, we labeled all pairs with a lower metric value as belonging to not guilty
and identified the largest value that meets 100 percent precision—no pair
classified as not guilty is misclassified.

\[
MLC(A,B) = |a| + |b| - \text{common substring length}
\]

For these 50 products, we inspected

1,225 product pairs \((50C_2)\) to determine if each pair included reuse-based
clones. Of the clone pairs detected, we

inspected only those greater than or equal to 30 tokens, a common detection
threshold for token-based clone detectors.\(^3\) Also, as we show shortly, 30
tokens is a small enough value to identify an upper bound with which
to arrive at a guilty verdict. (Although technically no pairs can violate OSS
licenses because all are LGPL or GPL; for illustration, we treat them as if they
could be guilty in terms of whether or not they contain reused code.)

\[
NCP = \min\{a, b\}
\]

For each pair, an expert programmer
(one of us) picked the largest clone
and judged on the basis of his experience if it was likely to be produced
accidentally. If no, the clone was con-
sidered reuse-based (guilty); if yes, it
was deemed accidental (not guilty).

Similarly, to identify each metric’s upper
bound, we labeled all pairs with a lower metric value as belonging to not guilty
and identified the largest value that
meets 100 percent precision—no pair
classified as not guilty is misclassified.
Results

Figure 2 shows the resulting MLC values. Figure 2a shows precision and recall for identifying guilty products by MLC. We identified an MLC of 305 as the lower bound for a guilty verdict—that is, all product pairs are guilty if they contain clones greater than or equal to 305 tokens. At this point, recall is 75.2 percent, which means that relying solely on the MLC metric overlooks 24.8 percent of the guilty pairs. As Figure 2b shows, we identified an MLC of 58 as the upper bound for a not guilty verdict—that is, all pairs are not guilty if they contain clones less than or equal to 58 tokens. At this point, recall is 84.8 percent, which means 15.2 percent of the not guilty pairs are being overlooked. The rest of the product pairs—those with an MLC value of 59 to 304, which constitute 16.1 percent of all the pairs—remain suspicious, needing human inspection for a conclusive verdict.

As Figure 3 shows, we found neither an upper bound NCP nor a lower bound NCP for distinguishing accidental and reuse-based clones. Our finding implies that clone pairs of 30 tokens exist everywhere regardless of whether or not they contain reused code. This result confirmed, somewhat
to our surprise, that NCP is not a useful metric with which to identify reuse-based clones, although many people consider it an effective measure.

Figure 4 gives the resulting values for LSim. As Figure 4a shows, we identified a lower bound of 25.2 percent for a guilty product—that is, a pair of source files with the largest clone is guilty if its clone coverage (within a file) is greater than 25.2 percent. However, because recall is 76 percent at this point, 24 percent of the guilty pairs are being overlooked. As Figure 4b shows, we identified an upper bound of 2.79 percent for a not guilty product. At this point, recall is 91.1 percent, which means that 8.9 percent of the not guilty pairs are being overlooked. The rest of the product pairs—those with an LSim value of 2.8 to 25.1 percent, which is 10.4 percent of all pairs—remain suspicious, needing human inspection for a conclusive verdict.

These results aren’t without limitations, the foremost being that we inspected only 50 OSS products. Others must take care in using our boundary values, although we believe that these values can certainly lead to the correct identification of reuse-based OSS code in most products.

Another limitation of our results is that Type 2 clones are susceptible to program-transforming attacks. Future work might use Type 3 clones or some other plagiarism-detection method such as software birthmarks, which can better withstand program transformation.

Finally, our results did rely heavily on our expert programmer’s initial categorization. However, we believe that the programmer has sufficient experience reading and writing a variety of software to make his judgment reliable. Moreover, for many product pairs, there is no other realistic way to distinguish reuse-based and accidentally produced clones. Such judgment is both difficult and time-consuming, and we know of no conventional study that has tried to identify the upper and lower bounds of clone metrics that can reveal guilty and not guilty products.

**Classification Performance**

Our results show that both MLC and LSim are effective metrics in identifying OSS code reuse. Both metrics are also effective as predictor variables to classify guilty/not guilty product pairs through a logistic regression model—a common modeling technique for two-group classification problems. To evaluate the model’s classification performance, we used 100 repetitions of twofold cross validation, in which we randomly took two-thirds of the entire dataset to build a model and used the remaining third for performance evaluation.
Figure 5 shows the resulting box plots of precision and recall for identifying guilty products. The average precision was 0.955, which indicates very low (4.5 percent) misclassification of guilty products. Similarly, the average recall was 0.871, which indicates that this classification overlooked a reasonably low percentage (12.9 percent) of guilty products. In addition, the average false-positive rate (ratio of misclassified not guilty products to total not guilty products) was extremely low (0.51 percent). Thus, MLC and LSim together can result in a high precision of guilty product identification with relatively low false negative error and very low false-positive error.

We identified and experimentally tested clone-based metrics and their threshold values for distinguishing accidental and reused-based OSS code. Through this work, we were able to ascertain that MLC and LSim are the most effective clone metrics in identifying software reuse and that both MLC and LSim had both a lower-bound threshold to identify guilty products and an upper-bound threshold to identify not guilty products. NCP, on the other hand, wasn’t effective in distinguishing guilty or not guilty products.

Our logistic regression model for classifying guilty products using MLC and LSim together correctly identified 95.5 percent of guilty products, while overlooking only 12.9 percent of guilty products on average. We believe that our results offer a viable alternative to existing detection services and a low-cost, relatively painless way to avoid reuse violations.

References