A Code Clone Oracle

Daniel E. Krutz and Wei Le
Software Engineering Department
Rochester Institute of Technology, USA
1 Lomb Memorial Drive
Rochester, NY 14623
{dxkvse,wei.le}@rit.edu

ABSTRACT

Code clones are functionally equivalent code segments. Identifying code clones is very important for determining bugs, fixes and software reuse. Code clone detection is also essential for developing fast and precise code search techniques. *[Dan says: I am a bit confused by this sentence, maybe reword it]* However, the challenge of such research is to evaluate that the clones detected are actually functionally equivalent, considering the majority of clones are not textual or even syntactically identical. In fact, we found that different tools report inconsistent results regarding what is a clone. The goal of this work is to generate a set of high-confident code clones to help future code clone detection and code search tools to evaluate their techniques. We studied three open source programs, *Apache*, *Python* and *PostgreSQL*, and randomly sampled a total of 1536 function pairs. We first applied multiple code clone detection tools to identify code clones. Meanwhile, we built a tool to automatically load function pairs of interest and record the manual inspection results on whether or not the two functions are clones. We recruited three *experts* who have experience in clone detection research and four students who have experience in programming to manually determine code clones and their types. We found 66 pairs of clone functions that agreed by both tools and human inspection results, which can be used as an oracle for future researchers to determine whether a tool successfully identified code clones and their types. The 66 pairs cover all 4 types of clones. Interestingly, only 1 pair is *type 1*, which is textual identical, and 9 pairs are *type 4* clones, which only can be determined by comparing input and output of the functions.

1. INTRODUCTION

Code clones are multiple fragments of source code that may be syntactically different, but are functionally equivalent. Code clones may adversely affect software in a variety of ways. Some of which include elevating the maintenance costs of a project, since the same alterations may need to be done several times, and inconsistent bug fixes leading to increased system faults [8].

Clones have been classified into four primary groups. Type-1 clones are the simplest and represent identical code except for variations in whitespace, comments and layout. Type-2 clones are syntactically equivalent except for differences in identifiers and types. Type-3 clones are two segments which differ due to altered or removed statements. Type-4 clones are the most difficult to detect and represent two code segments which considerably vary syntactically but produce identical results when executed [8].

Numerous tools and techniques have been proposed to discover code clones using a variety of different and highly effective methods [8]. When measuring the effectiveness of new or existing tools in terms of precision, recall, or in their ability to discover different types of clones, they are often evaluated against clone oracles [2, 5]. These oracles are usually small classes, containing only 5-20 known clones, with the tools being evaluated on how well they can find the known clones and not identify false positives [3].

Some oracles are created exclusively by manually analyzing applications for clones. This is a difficult and time consuming task since it is very hard to manually identify more complicated varieties of clones, such as type-4 clones. Additionally, manually identifying clones is a difficult and imprecise task even for researchers very experienced with clone detection [11]. In other instances, oracles are created through the use of a single clone detection tool, which creates a bias towards the tool and the clones it is able to find.

In the following work, we present a publicly available code clone oracle which is substantially sized, and has been generated through a mixture of manual analysis and several leading clone detection tools. To create this oracle, we first selected several well known open source applications and had researchers and students independently examine them for clones. Next, several leading clone detection tools were run on this code base. The results from these tools were mixed with the findings from the manual analysis phase to determine the clones in each system, in addition to the type of clone found. A separate student group also provided a separate rating of the clone pairs.

This paper is innovative because there are no known existing clone oracles which are substantially sized, we explicitly define all four types of clones, and the oracle has been created through a mixture of manual verification and several leading clone detection tools to eliminate as much potential bias as reasonably possible.

The rest of the paper is organized as follows. Section 2

MSR '14 Hyderabad, India

describes how the oracle was created and some of the contents from the oracle including the amount of each type of clone discovered and how many functions in each application were not clones. Section 3 provides an overview of related works, while Section 4 discusses future work to be done on this area. Finally, Section 5 provides concluding remarks to this work.

2. ORACLE CREATION

In the following sections, we describe our selection of the source code of several applications, and our identification of code clone candidates through the use of manual analysis and with the assistance of several leading clone detection tools. We also discuss how our data is made available to other researchers for use in their own work.

2.1 Clone Identification

The initial step in the oracle creation process was to select several existing applications to examine for clones. We chose Apache 2.2.14[1], Python 2.5.1[2] and PostgreSQL 8.5[3] since they are significantly sized, well known applications which have already been used in previous clone detection research [2]. These were analyzed as-is and had no alterations performed to their source code. For this analysis, we decided to search for clones at the method level. This precision was chosen since manually analyzing code at a more fine-grain level would not have been reasonably possible due to the immense amount of manual effort required by this task. The examined source code is available on the project website.

Two separate groups manually analyzed the possible clone pairs. One was three researchers very familiar with code clones, which we will refer to as our clone experts, and four students who also analyzed the candidate clone pairs. While the expert group discussed their findings to come to a consensus, we merely report the percentage of students who identified if a candidate pair represented a clone. The results from the two groups are shown independently of each other. Our goal for having these two groups was to present more information for future researchers to consider, and use if they should desire. A secondary goal was to see how much the findings of the expert and student groups differed from one another.

The four types of clones as defined by Roy *et al.* [8] were used as guidance in identifying and classifying clones. In order to manually evaluate each of the systems for clones at the method level, all methods in the application would need to be compared against all other methods in the application in a round robin fashion, with the clones and their types being recorded. The number of comparisons done using this process was extremely large, as show in Formula 1, so even an application with 100 methods would need to have 4,950 comparisons performed.

$$(MethodCount * (MethodCount - 1))/2 \qquad (1)$$

To make this process more manageable, we randomly chose a small subset of 3-6 class files from each application to examine for clones. We arbitrarily selected several classes in each application for analysis, with the only selection criteria being that each class was to contain at least ten methods so that we would have an adequate number clone candidates. The number of possible clones in these subsets for the three applications was still over 45,000; which is not reasonable to check for through manual analysis. To make this number more manageable, we randomly selected a statistically significant number of clone possibilities to examine with the goal of having a confidence level of 99% and a confidence interval of 5.

We developed an open source tool known as *GraphicDiff* [4] to assist with the manual clone identification process. This tool automatically displayed each of the methods to be compared and allowed the user to select if the comparison represented a clone, and if so what type. Once the user was finished examining a candidate clone pair, they could then easily cycle onto the next candidate pair. Each research group independently examined the applications using GraphicDiff, with their findings recorded. GraphicDiff is represented in Figure 1.

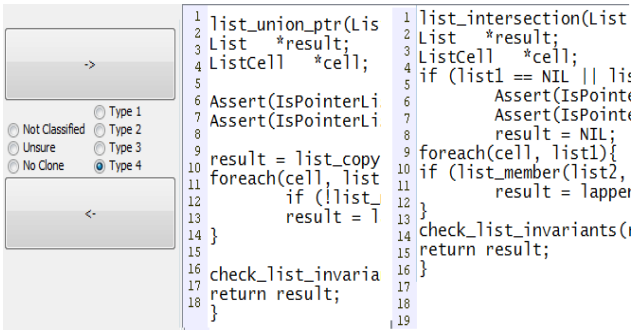

Figure 1: GraphicDiff

2.2 Expert Group

The clone identification process for our expert group was assisted by several leading clone detection tools including Simcad [10], Nicad [7], MeCC [2] and CCCD [4]. The results of each clone comparison for all of the tools were recorded. This step was done after manual analysis in order to eliminate any possible influence on the initial manual analysis phase.

There were numerous discrepancies between the evaluators during the manual analysis phase. In order to help mitigate these disagreements as to whether or not two methods were clones, or even what type of clone they represented, results from the various tools were used by the researchers to assist with the decision making process. Using these results as input, discrepancies were discussed until an agreement could be made. There were cases however, where no agreement could be made with these results being recorded as *unsure* in the final result set.

If several of the clone detection tools indicated a clone, where none was noted during manual analysis, the reviewers re-evaluated the candidate clone pair to ensure they had not overlooked anything. The same process was used in reverse when the tools indicated no clone, but manual analysis had found a clone pair. Ultimately, the final decision on a

[1]http://www.apache.org
[2]http://www.python.org
[3]http://www.postgresql.org

[4]https://code.google.com/p/dek-graphicdiff/

possible clone pair was made by the researchers, and not by any tool.

2.3 Student Group

A group of four students examined the same set of candidate clone pairs as the expert group to provide a larger, more diverse set of results for consideration. All were upper division software engineering students who had no prior experience with code clones. To help familiarize students with code clones, they were asked to read papers by Roy *et al.* [8], Kim *et al.* [2], and Lavoie and Merlo [5]. Finally, all students were independently interviewed to ensure that they understood code clones at an acceptable level.

The students did not discuss their results or come to a conclusion with other student examiners, and were not provided with any results from clone detection tools, or from the expert group. For each candidate clone pair, the percentage of students who identified it as a clone are presented for use at the discretion of the researcher. Like the expert group, the students used GraphicDiff to assist in the clone identification process.

2.4 Results

Our created oracle is available on our project website[5] in several formats including html, csv, xml and xls. Table 1 shows an example set of publicly available data. The comparison column lists the two methods which were evaluated. The expert column displays the type of clone agreed upon, and a "No" if a clone was not found. The student column shows the percentage of students who found the pair to represent a clone. Finally, subsequent columns list the tools used as part of the analysis and if they found the pair to represent a clone. Since only a subset of comparisons were manually examined by either of the two human groups, the comparisons which were not manually analyzed were left blank.

Table 1: Example Results Output

Comparison	Expert	Student	Tool 1	Tool 2
MethA-MethB	Type 1	100	No	Yes
MethB-MethC			No	No
MethA-MethB	Type 3	75	Yes	Yes
MethD-MethE	No	0	No	No

Table 2 displays the various types of classifications as noted by our *expert* and *student* groups. The clones were broken down by type for the expert group, while the student group was not since they were only asked to identify the clones, not classify them. The relatively small number of type-4 clones identified in the oracle is because even the best clone detection tools struggle at finding these most complicated types of clones and identifying them through manual analysis is a very difficult task due to their complexity [8]. No type-1 clones were noted, which is not surprising since developers can usually recognize exact duplications of code and would have removed them from the software. All of the type-4 clones were initially manually classified as type-3 clones. Only after a subsequent discussion among the researchers did they conclude that they were actually type-4 clones.

[Dan says: I left the section about total clones found by 50% of the student researchers. Feel free to remove it.]

The number of clones discovered for each group of evaluators significantly differed. While there were only 5 more clones identified by the student group in Apache, this number grew to 19 in Python and 74 in PostgreSQL. There are several reasons for these variations. The first is that we only used 50% as the identification criteria for the student group and reducing this value would have removed many of the clones identified by the students. However, we wanted to present these findings for future researchers to use at their discretion and therefore decided to keep the classification criteria at 50%. Additionally, there is expected to be a large amount of variation for the manual classification of clones, so a significant about of deviation is expected [11]. The *Agree* column is the number of clones identified by the expert group, that were identified as clones by at least 50% of the student group. The students agreed with the majority of clones in Python and P-SQL, but only 7 of the 18 clones in Apache.

[Dan says: Should we attempt to explain why? I don't really have a firm reason.]

Table 2: Clones Identified in Oracle

Application	Clone	Expert	Student	Agree
Apache	**T1**	0		
	T2	18		
	T3	0		
	T4	0		
	Total	18	23	7
	Not Clone	339	334	303
Python	**T1**	0		
	T2	7		
	T3	4		
	T4	4		
	Total	15	34	12
	Not Clone	530	522	473
P-SQL	**T1**	0		
	T2	18		
	T3	10		
	T4	5		
	Total	33	107	33
	Not Clone	601	550	459
Total	**Clone**	66	164	46
	Not Clone	1470	1406	1208

A sample clone from PostgreSQL is shown in Listing 1 and Listing 2. This was identified as a type-4 clone after a brief discussion by the expert group, while 50% of students classified this as a clone.

Read more: http://www.physicsforums.com

[What else should]

Listing 1: Clone Example #1

```
list_union_ptr(List *list1, List *list2)
List    *result;
ListCell    *cell;

Assert(IsPointerList(list1));
Assert(IsPointerList(list2));

result = list_copy(list1);
foreach(cell, list2){
    if (!list_member_ptr(result, lfirst(cell)))
    result = lappend(result, lfirst(cell));
}

check_list_invariants(result);
return result;
```

Listing 2: Clone Example #2

```
list_intersection(List *list1, List *list2)
List    *result;
ListCell    *cell;

if (list1 == NIL || list2 == NIL)
return NIL;

Assert(IsPointerList(list1));
Assert(IsPointerList(list2));

result = NIL;
foreach(cell, list1){
    if (list_member(list2, lfirst(cell)))
    result = lappend(result, lfirst(cell));
}

check_list_invariants(result);
```

[5] http://www.se.rit.edu/~dkrutz/cloneoracle/

Table 3: Summary of our Findings

Research Questions	Comparison Metric	Automatic	Manual	Implications
Why we group crashes	Grouping criteria	Single: same cause	Multiple: same, related causes, who should fix	Exploit the uses of groups to better prioritize and fix bugs
How we group them	Grouping info	Limited: signatures, call stacks	Multi-sources: black-box + white-box info	Correlate multi-sources info, multi-versions, multi-applications
	Imprecision	Fundamental: based on symptoms	Ad-hoc: incorrectly parse and link info	Design better tools to reveal relations between symptoms and code
What are the capabilities	Call stack characteristics	Scalable: larger, more call stacks	Diverse: dissimilar between call stacks	Grouping based on call stacks is insufficient, especially for small apps

Code Segment #1

```
list_union_ptr(List *list1, List *list2)
List   *result;
ListCell   *cell;

Assert(IsPointerList(list1));
Assert(IsPointerList(list2));

result = list_copy(list1);
foreach(cell, list2){
    if (!list_member_ptr(result, lfirst(cell)))
    result = lappend(result, lfirst(cell));
}

check_list_invariants(result);
return result;
}
```

Code Segment #2

```
list_intersection(List *list1, List *list2)
List   *result;
ListCell   *cell;

if (list1 == NIL || list2 == NIL)
return NIL;

Assert(IsPointerList(list1));
Assert(IsPointerList(list2));

result = NIL;
foreach(cell, list1){
    if (list_member(list2, lfirst(cell)))
    result = lappend(result, lfirst(cell));
}

check_list_invariants(result);
return result;
}
```

I say about this?]

3. RELATED WORKS

There have been other oracles created to assist in the evaluation of clone detection tools. Krawitz [3] and Roy *et al.* [8] both explicitly defined clones of all four types in a small controlled environment. However, these works only specified a small number of clones which were artificially created.

In 2002, Bailey and Burd [1] formed a human clone oracle which was used to compare three of the leading clone techniques at the time. While this oracle has been used many times in evaluating clone detection tools, it has been criticized due to its validation subjectivity and its relatively small size. Bellon *et al.* created a large human validated oracle. However, only a small number of discovered clones were oracled and the results may have been adversely affected by other limitations [8].

Works by Li and Ernst [6] and Saebjornsen *et al.* [9] created oracles using clones as identified by software developers. However, it is possible that developers only reported a portion of the clones in the system which is extremely likely given the difficulties in manually identifying many types of clones [11] and could lead to a large number of false positives. Lavoie and Merlo [5] described an automated method of building an oracle containing type-3 clones on massive data sets which was based on the Levenshtein metric. While this is a powerful, automated technique for producing clones in large data sets, this is the only process used to create the oracle and does not contain any manual verification. Like with the other proposed oracles, it does not contain any explicitly defined type-4 clones.

4. DISCUSSION & FUTURE WORK

While we feel that we have created a robust and useful clone oracle, there are improvements and future work which may be done. Even though we had several developers experienced with clones work to identify and classify potential clones, we are sure that there are possible inaccuracies and decisions which other researchers may disagree with. While clone identification and classification is a difficult task and one which is likely to lead to disagreements even in a perfect scenario [11], more researchers would likely lead to more accurate and externally agreed upon results.

We used a wide variety of leading clone detection tools to help guide our decision making process. However, the use of more clone detection tools would likely be able to assist with this process. Additionally, while there are many other

robust clone detection tools, it is unreasonable to use all them.

The oracle we created only identified clones at the method level. While many clone detection tools are capable of only identifying clones at the method level, others may find them at a more granular level [8]. Work may be done to create an oracle at the sub method level. However, initially identifying clones at the sub method level is largely a very unreasonable task due to the overwhelming amount of manual effort that would be required. We believe our oracle is still more than sufficient to assist with the evaluation of new and existing clone detection tools.

Finally, our oracle is only C-based, meaning that the it will be of no use to clone detection tools which analyze code of other languages. Our technique is still very useful since a large portion of existing detection tools are C-based [8] and our intention was never to create a large and robust oracle for every conceivable language.

5. CONCLUSION

This work described the creation of a large and robust code clone oracle which contains explicit examples of all four types of clones. The oracle was created through a mixture of manual analysis, and results from a variety of clone detection tools. The results are publicly available in a variety of formats on the project website and is encouraged for use by future code clone researchers.

6. REFERENCES

[1] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 36–, Washington, DC, USA, 2002. IEEE Computer Society.

[2] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[3] R. M. Krawitz. *Code Clone Discovery Based on Functional Behavior*. PhD thesis, Nova Southeastern University, 2012.

[4] D. E. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, 2013.

[5] T. Lavoie and E. Merlo. Automated type-3 clone oracle using levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 34–40, New York, NY, USA, 2011. ACM.

[6] J. Li and M. D. Ernst. Cbcd: cloned buggy code detector. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 310–320, Piscataway, NJ, USA, 2012. IEEE Press.

[7] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[8] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[9] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[10] M. Uddin, C. Roy, and K. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238, 2013.

[11] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 285–, Washington, DC, USA, 2003. IEEE Computer Society.