Examining the Effectiveness of Using Concolic Analysis to Detect Code Clones

give the original reference for the type-X definitions: Comparison and Evaluation of Clone Detection Tools by Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, Ettore Merlo IEEE Transactions on Software Engineering Vol. 33, No. 9, pages 577-591, 2007

Daniel E. Krutz and Emad Shihab
Rochester Institute of Technology
{dxkvse,emad.shihab}@rit.edu

***Abstract*—Software is often expensive to build and maintain. During the initial construction and subsequent maintenance of an application, functionality is often duplicated throughout the system. This redundancy may occur for numerous reasons and has several adverse effects on a software project. Some include an increased size of the codebase, elevated maintenance costs and inconsistent developer changes due to heightened program comprehension needs. This replicated functionality is often known as a code clone. A code clone is defined as multiple code fragments that produce similar results when provided the same input.**

This paper presents a new process for detecting code clones based on concolic analysis. This process finds clones by performing concolic analysis on the targeted source code and then examining the concolic output for similarities. Analogous concolic output is indicative of a code clone candidate. To evaluate the effectiveness of the proposed process, we compared its performance to leading clone detection tools. This comparison was done using both clones from existing research benchmarks as well as examining several open source applications for clones using these tools.

In our study, concolic analysis was able to detect 90% of the known clones in three open source applications. Additionally, concolic analysis was able to detect and document 393 more clones in the same portion of the application as compared to a leading clone detection tool. In a controlled environment, concolic analysis found 50% more clones than the next closest clone detection tool.

I. Introduction

Software must continually change in order to keep up with user requirements. These alterations may also be done to enhance the functionality of the software, fix bugs, or repair security vulnerabilities. Prior work showed that in many cases, code changes contain cloned code. Code clones occur in software for a variety of reasons. One reason is that developers may knowingly duplicate functionality across the software system. This may be due to an unwillingness to refactor and retest the modified portion of the application, or simply due to laziness on the part of the developer. In other instances, developers may not be aware that identical functionality exists in the system and may unknowingly inject clones into the application [2], [12], [22], [31]. Clones continue to be extremely widespread in software development. It is estimated that clones typically comprise between 5-30% of an application's source code [5] [40] [25].

A significant amount of previous work states that code clones are undesirable [32] [12] [2] [5]. Clones may lead to more bugs in a system and may make the defect remediation process more difficult and expensive. Clones also substantially raise the maintenance costs associated with an application [21]. Reducing maintenance costs is a very serious matter since the maintenance phase of a project has been found to typically comprise between 40-90% of the cost of a software project [43] [13] [41] [46] [6] [14]. Finally, unintentionally making inconsistent bug fixes to cloned code across a software system is also likely lead to further system faults [10].

There are four types of code clones which are generally recognized by the research community. Type-1 clones are the simplest and represent identical code except for variations in whitespace, comments and layout [48]. Type-2 clones are syntactically similar except for variations 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 ~~represents~~ represent two code segments which differ considerably syntactically, but produce identical results when executed [9], [16].

To assist software practitioners in detecting and managing code clones, clone detection tools have been proposed. Such code clone tools have helped in detecting clone-related bugs and even security vulnerabilities in software systems [9]. There are numerous clone detection tools which detect different types of clones.

Many of the existing clone tools are able to detect the simpler clones of the type-1, type-2 and type-3 classification. Type-4 clones are typically the most difficult clones to detect and may be more problematic than the more basic clone types [49] [36]. The detection of type-4 clones is a critical, yet difficult, task. To the best of our knowledge, only two known processes are able to detect the most complicated types of clones, type-4 [27], [35]. MeCC is the only known implemented tool which is capable of reliably detecting type-4 clones. This tool suffers from several drawbacks including the ability to only ~~analyzed~~ analyze pre-processed C program and an excessive clone detection time which is likely caused by the exploration of an unreasonably large number of possible program paths [24] .

Ref. 35 does not introduce a type-4 detection

In this paper, we examine the effectiveness of using concolic analysis to detect code clones. Concolic analysis combines concrete and symbolic values in order to traverse all possible paths of an application (up to a given length). Concolic analysis has been traditionally used in software testing to find application faults [23], [26]. Concolic analysis forms the basis of a powerful clone detection tool because it only considers the functionality of the source code and not its syntactic properties. This means that things such as comments and naming conventions which have been problematic for existing clone detection systems will not affect concolic analysis and its detection of clones.

This clone detection technique is innovative for several reasons. First, only two other known works are able to detect type-4 clones. Additionally, it represents the only known process for detecting clones which is based upon concolic analysis.

Our study aims to answer the following research questions:

RQ1: *What types of clones is concolic analysis effective at detecting?*
We found concolic analysis to be capable of detecting all types of clones in both a controlled environment and in several open source applications. These results were manually verified by several researchers. In a controlled environment, concolic analysis was able to detect 100% of type-1 and type-2 clones, 93% of type-3 clones and 83% of type-4 clones.

RQ2: *How does concolic analysis based clone detection compare to other leading clone detection tools?*
While several existing methods are very innovative and successful at detecting a variety of code clones, we found that concolic analysis compares very favorably to these tools. Concolic analysis was capable of finding 90% (657/733) of the clones as found by several existing tools, along with detecting an additional 393 clones in portions of several open source applications which were not detected by leading tools.

The remainder of the paper is organized as follows. Section II describes how concolic analysis may be used to detect software clones. Section III provides an example of why finding clones using concolic analysis is important. Section IV evaluates the ability of concolic analysis in identifying clones in relation to existing tools. Section V conveys interesting results from the research. Section VI discusses related works in clone detection and concolic analysis. Section VII provides concluding remarks and future research directions for this work.

II. How Concolic Clone Detection Works

A. Concolic Clone Detection Process

In order to explain ~~of~~ how concolic code clone detection works, we will first provide an example of two code clones and then describe how concolic analysis is able to detect these clones. Two type-2 clones are shown in Listing 1 and Listing 2. These are derived from clones presented by Roy *et al.* [35].

Concolic code clone detection is comprised of two primary phases. The first step is the generation of the concolic output on the target application. This may be done using an existing concolic analysis tool such as CREST[1], Java Path Finder (JPF)[2], or CATG[3].

[1]http://code.google.com/p/crest/
[2]http://babelfish.arc.nasa.gov/trac/jpf/
[3]https://github.com/ksen007/janala2

```
void sumProd(int n) {
  double sum=0.0;
  double prod =1.0;
  int i;
  for (i=1; i<=n; i++){
    sum=sum + i;
    prod = prod * i;
    foo2(sum, prod);
  }
  }
```

Listing 1. Clone Example #1

```
void sumProd2(int n) {
    int sum=0; //C1
    int prod =1;
    int i;
    for (i=1; i<=n; i++){
      sum=sum + i;
      prod = prod * i;
      foo2(sum, prod);
    }
  }
```

Listing 2. Clone Example #2

An abbreviated example segment of concolic output is shown in Listing 3. The complete portion is not shown $\overset{\text{due}}{\text{do}}$ to space limitations, but may be viewed on the project website http://www.se.rit.edu/~dkrutz/cccd/. The generated concolic output represents all executable paths which the software may take. The output is broken into several *path conditions*. These are conditions which must be true in order for the application to follow a specified path. For example, if in order to follow a specific path of an *if* statement, a boolean variable needed to be *true*, the ~~a~~ contingency of the path condition would be that the variable be *true*. Otherwise, this path will not be traversed [42].

```
PC#=3
CONST_3>a_1_SYMINT[2]&&
CONST_2<=a_1_SYMINT[2]&&
CONST_1<=a_1_SYMINT[2]
SPC#=0

PC#=2
CONST_2>a_1_SYMINT[1]&&
CONST_1<=a_1_SYMINT[1]
SPC#=0

PC#=1
CONST_1>a_1_SYMINT[2]
SPC#=0
```

I don't understand this. Please explain.

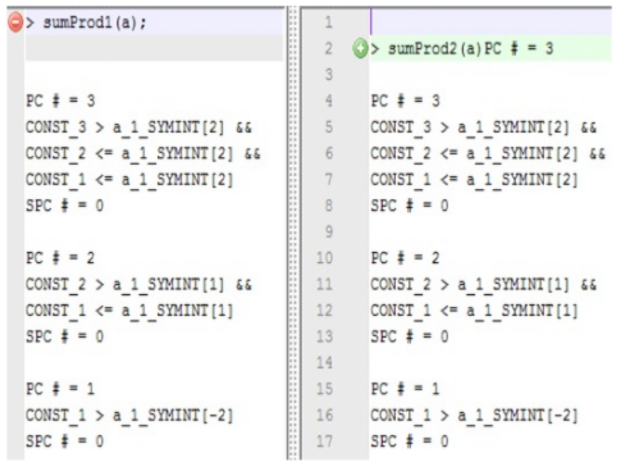
Listing 3. Example Concolic Output

In the abbreviated example above, constant variable types are represented generically by "CONST" while the variable type integer is represented by a generic tag "SYMINT." Various other variable types are represented in a similar fashion.

Actual variable names do not appear anywhere in this concolic output and are irrelevant to the concolic analysis process. When comparing the concolic output from the type-2 clones in Listing 1 and Listing 2, one of the primary differences would be that *sum* would be defined as a double variable type for the first method while it would be an int for the second. This would create a small variation in the compared output.

Once this concolic output is created, similar sections of output will be searched for and will be noted to be code clone candidates. Concolic analysis explores the possible paths that an application can take. Similar application execution paths signify analogous functionality and is thus indicative of a code clone candidate. what about different cfg that still compute the same output?

In order to demonstrate this functionality, clones as defined by Roy *et al.* [35] in Listing 1 and Listing 2 were analyzed using concolic analysis. The generated concolic output was compared for variations. The only difference between the two sets of concolic output is the method name displayed at the top of each file. An abbreviated listing is shown in Figure 1 with full results is available on the project website.



```
> sumProd1(a);                     
                                   > sumProd2(a)PC # = 3

PC # = 3                           PC # = 3
CONST_3 > a_1_SYMINT[2] &&         CONST_3 > a_1_SYMINT[2] &&
CONST_2 <= a_1_SYMINT[2] &&        CONST_2 <= a_1_SYMINT[2] &&
CONST_1 <= a_1_SYMINT[2]           CONST_1 <= a_1_SYMINT[2]
SPC # = 0                          SPC # = 0

PC # = 2                           PC # = 2
CONST_2 > a_1_SYMINT[1] &&         CONST_2 > a_1_SYMINT[1] &&
CONST_1 <= a_1_SYMINT[1]           CONST_1 <= a_1_SYMINT[1]
SPC # = 0                          SPC # = 0

PC # = 1                           PC # = 1
CONST_1 > a_1_SYMINT[-2]           CONST_1 > a_1_SYMINT[-2]
SPC # = 0                          SPC # = 0
```

Fig. 1. Diff of Concolic Output

In order to measure the similarity between sets of the concolic output, the Levenshtein distance measurement is used. Levenshtein distance is defined as the minimal number of characters that would need to be replaced in order to convert one string to another [3] [17]. For example, if string #1 was "ABCD" and string #2 was "BCDE", then these two values would have a Levenshtein distance of 2. This is because the character A would need to be removed from string #1 and E would need to be removed from string #2. This string similarity technique was selected for several reasons. Other string similarity techniques would prove to be impractical for use in clone detection using concolic analysis. For example, the Hamming technique may only be used with strings which are the same length [19], [34]. When comparing the concolic output of even two very similar methods, the length is not expected to be the same. The longest common subsequence technique does not account for the substitution of values, only the addition and deletion of characters [30].

Is Levenshtein distance appropriate? It is a syntactic measure (for sequences), while the concolic output are semantic properties.

While there are other string similarity techniques which may be viable options, the Levenshtein distance metric was found to be well suited for this clone detection task. This measurement technique is amiable to the process of measuring the distance between segments of code due to its ability to work with strings of different lengths and its restriction of upper and lower bounds in the calculated distances. The Levenstein edit distance has also been used in previous work comparing code in version repositories [7] and in measuring the distance between call stacks [4]. In order to help normalize the similarity results based on length, the final Levenshtein score (ALV) is computed by dividing the Levenshtein distance between two files (LD) by the longest string length of the two strings being compared (LSL) and then multiplying by 100. The ALV formula is shown below in Equation 1.

$$ALV = (LD/LSL) \times 100 \qquad (1)$$

The basic technique of detecting clones using concolic analysis is language agnostic. Any existing concolic analysis tool may be used in this process. In order to evaluate the effectiveness of concolic analysis in detecting clones, two implementations were created. The first was a tool which utilized CREST to generate the necessary concolic output and was able to analyze source code written in C. An overview of the entire clone detection process is shown in Figure 2.

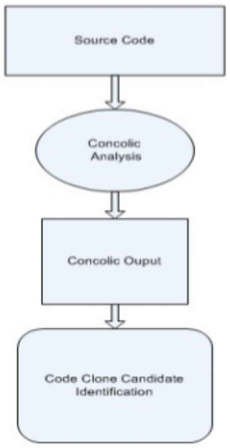


Fig. 2. Concolic Analysis Process

A prototype was also created using JPF as the driving force for generic concolic analysis and was able to examine source code written in Java. While not nearly as robust as the C-based implementation using CREST, this Java based prototype was important because it demonstrated several key concepts. While concolic analysis for clone detection has been implemented in several languages, the actual process is itself language agnostic. This Java based implementation was also created in order to demonstrate that the process was not only limited to the CREST tool.

In order to evaluate concolic analysis for clone detection, several leading existing clone detection tools were selected for comparison purposes. For Java these were CodePro [8] and CloneDR [11]. For C based applications, CloneDR and Simian [4] were chosen. Due to the magnitude of the codebase of the benchmark applications, random subsections were analyzed by concolic analysis and the other leading existing tools. A complete listing of results and program output may be found on the project website.

B. Concolic Analysis Clone Detection Tool

Concolic Code Clone Detection (CCCD) is a fully functional tool which has been developed and uses concolic anaylsis for clone detection. Concolic analysis is performed on the target application using CREST. A Java component uses CTAGS[5] to break up the concolic output at the method level. Next, a comparison process uses the developed Levenshtein distance based measurement to evaluate the similarity between the concolic output files. A final report then displays the detected code clone candidates. This tool, installation instructions, and further details may be found on the project website.

III. Motivating Example

In order to compare existing clone detection tools, a simple comparison was conducted between concolic analysis for clone detection and leading existing tools. This was accomplished using a type-3 clone as defined by Roy *et al.* [35] and is shown in Listing 4 and Listing 5.

```
void sumProd1(int n) {
  double sum=0.0;
  double prod=1.0;
  int i;
  for (i=1; i<=n; i++){
    sum=sum + i;
    prod = prod * i;
    foo2(sum, prod);
  }
}
```

Listing 4. Clone Example #1

```
void sumProd3(int n) {
  double sum=0.0;
  double prod =1.0;
  int i;
  for (i=1; i<=n; i++){
    if (i %2 == 0){
      sum+= i;
    }
    prod = prod * i;
    foo2(sum, prod);
  }
}
```

Listing 5. Clone Example #2

Along with concolic analysis for clone detection, several leading clone detection tools analyzed this example clone. These included CloneDR, Simian, CodePro and MeCC [24]. Concolic analysis was the only clone detection tool which was able to detect this type-3 clone. Because it is simply too small?

No! CloneDR is syntax-based not text-based!

CloneDR and Simian struggle at locating this clone because they are text based clone detection systems. Even though the two examined methods are functionally equivalent, they are syntactically different and therefore cause a problem for these text based tools. MeCC is a semantic based clone detection tool which means it detects clones using a static program analysis to generate more precise information than simply using syntactic similarities. While difficult to pinpoint the precise reason why MeCC was unable to detect this clone, it is likely due to the inability of the employed semantics based static analysis technique to generate equivalent abstract memory states for the two functionally equivalent methods. CodePro is a closed source application and makes no mention of what type of technique it uses for clone detection [20], so it is undecipherable why it is unable to detect this clone.

Concolic analysis was able to detect this clone because it only analyzes the functional nature of the software. The syntactic variations of the two methods which caused problems for the text based clone detection tools do not adversely affect the ability of concolic analysis to detect clones. The issues that hindered MeCC do not affect the ability of concolic analysis to detect clones in this situation because the concolic based technique does not use static analysis to explore functional application paths. Really? What is symbolic execution after all? It is a type of static analysis!

IV. Evaluation

In the following sections, concolic analysis for clone detection will be evaluated individually and against other leading existing clone detection tools. Since the Java based solution of concolic analysis for clone detection was a prototype, this implementation will only examine an initial proof of concept class and will not analyze the more complicated open source applications due to technical limitations of this prototype.

Several existing open source applications were selected for examination by concolic analysis as well as several leading existing clone detection tools. The chosen software was Apache 2.2.14 [6], Python 2.5.1 [7] and PostgreSQL 8.5 [8]. These applications and versions were selected because a substantial amount of benchmark information for MeCC was available from previous research on these software versions [1].

Due to the magnitude of the applications being analyzed, it was practically impossible to examine all of the identified code clone candidates since there are thousands of identified methods in each application. In order to make the evaluation process manageable, a subsection of each application was selected. A statistically significant portion of results were chosen for further analysis. Enough clone candidates were examined to provide a confidence level of at least 99% with a confidence interval of 10. Apache had 4,926 clone candidates identified by concolic analysis so 161 clones were chosen

=> clones are identified by the own analysis; bias towards own tool; no assessment of recall

[4] http://www.harukizaemon.com/simian/
[5] http://ctags.sourceforge.net

[6] http://www.apache.org
[7] http://www.python.org
[8] http://www.postgresql.org

what is the precision?

Analysis is limited to function clones. Say so early!

From which set of clones, clones were selected? Were these clones found by CCCD? This would create a major bias!

What are your criteria on when a pair of functions are clones? How were the clone candidates assessed? What measures were taken for quality control in this assessment?

for manual analysis. Python had 7,737 candidates so 163 were chosen for manual analysis while postgreSQL had 4,238 candidates of which 116 were manually analyzed. As time allowed, more clones from each application were randomly chosen for manual analysis. To help ensure accurate results, at least two researchers independently verified the results and discussed any discrepancies in order to come to an agreed conclusion. A distance metric of 40 was selected to indicate code clone candidates.

Next, we present our experiment results and answer our research questions.

RQ1: *What types of clones is concolic analysis effective at detecting?* This initial step of evaluating concolic analysis for clone detection was to verify if it was capable of detecting several clones of all types as defined by Krawitz [27] and Roy *et al.* [35]. These defined clones were then inserted into ~~into~~ a single Java and C file. Each of the tools then analyzed this file and searched for clones.

Language	Tool	T1	T2	T3	T4	Total
Java	CloneDR	5	4	0	0	9 (38%)
	CodePro	2	3	4	2	11 (46%)
	Concolic	5	6	6	6	23 (96%)
C	CloneDR	5	4	0	0	9 (38%)
	Simian	3	2	2	1	8 (33%)
	Concolic	5	6	7	4	22 (92%)
Total Possible		5	6	7	6	24

TABLE I
COMPARISON OF TOOLS ON SINGLE CLASS

The results are shown in Table I. For the Java implementation, the concolic analysis process was able to detect 96% of all clones with CodePro trailing with 46% and CloneDR only detecting 38%. The only clone which concolic analysis was unable to detect was a type-3 clone as defined by Roy *et al.*. The reason that the concolic analysis based clone detection was unable to detect that clone is because JPF was unable to traverse all paths of this method for technical reasons and was therefore unable to produce adequate concolic output. This is a limitation of JPF that ultimately effects the concolic analysis clone identification process. In the future, we plan to further enhance the concolic analysis clone detection process by using more advanced tools.

A similar C file containing the clones of Krawitz and Roy*et al.* was then examined for clones. As shown in the bottom half of Table I, concolic analysis was able to detect 92% of all clones while CloneDR had a detection rate of 38% with Simian trailing with 33%. Simian was able to detect one of the six injected type-4 clones in the application with CloneDr unable to find any.

The only clones that concolic analysis was unable to detect were the type-4 clones as defined by Krawitz. In this clone example, a method has been refactored into two functionally methods. Two different concolic paths were generated for these methods, and thus the generated concolic output was not similar, so no clone code candidate was detected.

RQ2: *How does concolic analysis based clone detection compare to other leading clone detection tools?*

Now that we have established that concolic analysis is effective at detecting code clones, we would like to examine how it compares to existing code clone tools. Due to the magnitude of code clone candidates identified by concolic analysis, a statistically significant number of clones were selected for analysis. These were randomly selected and enough clones were selected to ensure a confidence level of at least 99% and confidence interval of 10. This meant that at least 161 were selected for Apache, 163 for Python and 116 for PostgreSQL. repetition!

The results for this analysis are described in Table II. The number of clones found by concolic analysis which were not detected by the comparison clone detection tools are broken down into each type of clone. A complete listing of results may be found on the project website.

Application	Tool	T1	T2	T3	T4	Total
Apache	MeCC	8	22	70	0	100
PostgreSQL	MeCC	2	21	55	3	187
Python	MeCC	3	19	54	30	106
Total		x	x	x	x	x

missing totals

TABLE II
CLONES FOUND BY CONCOLIC ANALYSIS AND NOT OTHER TOOLS

This table represents the significant number of clones that concolic analysis was able to detect compared to existing tools. These results include a high number of type-4 clones, especially against the Python code base. These results demonstrate the effectiveness of concolic analysis in detecting clones which leading clone detection tools had a harder time detecting. but CCCD may have missed other clones

Clone detection results from MeCC were analyzed to see if concolic analysis was also able to detect these same clones. The clone detection technique based on concolic analysis was compared against results from MeCC. MeCC was selected as a comparison benchmark because it is a well known tool that it is able to detect type-4 clones and represents a semantic clone detection technique.

Due to the significant number of clones identified by all of these detection techniques, it was unreasonable to manually analyze all of them. Clones were selected and evaluated at random using a similar selection technique as described in RQ#1.

A complete listing of results is shown in Table III. The t~~T~~otal clones not found are represented in the column TCNF (Total Clones Not Found), while the total number of clones identified by the tool are displayed in the column TCI (Total Clones Identified).

While concolic analysis did not locate every clone as identified by existing tools, it did find a significant portion of them. Concolic analysis was able to recognize 162 of the 191 clone groups found by MeCC for a detection rate of about 85%. It is important to remember that concolic analysis was also able to find 100 more clones in the same application than MeCC was. These results improve for both PostgreSQL and Python.

No runtimes are given!

Language	Tool	T1	T2	T3	T4	TCNF	TCI
Apache	MeCC	0	12	14	2	29	191
PostgreSQL	MeCC	0	4	9	0	13	278
Python	MeCC	3	22	9	0	34	264

TABLE III
CLONES FOUND BY ~~OTHER TOOLS~~ AND NOT CONCOLIC ANALYSIS
MeCC

This result set is important because it demonstrates the ability of concolic analysis to detect a high rate of clones that were also found by existing tools.

V. DISCUSSION

A. The Impact of Levenshtein Distance

As stated earlier, one of the main parameters that needs to be set for the condolic analysis based clone detection is the similarity threshold, i.e., the Levenshtein distance. In all of our aforementioned experiments, we set the Levenshtein distance to 40. Now, we would like to examine the impact of varying the Levenshtein distance on our results.

To perform this comparison, we examine the number of clones in three open source applications - Apache, Python and PostgreSQL. For each project, we selected a statistically where from? significant portion of clone candidates and concolic analysis was performed on it with clone candidates being identified along with their calculated Levenshtein distance metric.

is LD calculated on the textual representation or token representation of the input parameter constraints?

Levenshtein distance	40	35	30	25	20	15
Not a Clone	342	200	124	77	45	30
Type 1	15	15	15	13	13	12
Type 2	74	70	67	60	47	41
Type 3	209	182	152	93	52	34
Type 4	34	28	22	17	9	7
Total Clones	332	295	256	183	121	94
Ratio	.97	1.48	2.06	2.38	2.69	3.13

only few type-4 clones

TABLE IV
CALCULATED LEVENSHTEIN SCORES

by what criteria?

Table IV shows our results. The number of candidates manually identified to not be clones are shown in the second row, called "Not a Clone". Next, we show the number of clones for each type (i.e., types 1-4). "Total Clones" is an aggregate true positives of all types of manually verified clones. The "Ratio" shows the devided by false positives? actual observed clones divided by the clone candidates which were actually not a clone. Each column of the table shows these number using different Levenshtein distances.

Several interesting findings resulted from this analysis. Even with the calculated Levenshtein score at its highest value (40), the ratio of clones compared to non-clones was essentially 1:1 with the ratio becoming much more favorable for accurate clone detection as the calculated Levenshtein scores are decreased. These detection ratios are very high for a clone detection tool. Recent research has shown that typically 75% of all clones as identified by leading tools represent false positives [45]. That said, we view the Levenshtein distance as an adjustable parameter that can be set to achieve a tradeoff between the number of clones detected and the number of false positives.

These results also demonstrate the ability of concolic analysis to detect all four types of clones in real world, open source projects. The detection of type-3 clones is something which few tools are able to reliably find, and only two known techniques are able to reliably detect type-4 clones with only one having been implemented into a functional tool.

B. Calculated Levenshtein Distance & Clone Types

An interesting finding is how the calculated Levenshtein distance may indicate the type of clone discovered. A lower calculated Levenshtein distance score is indicative of a closer level of similarity between two compared items. On average, type-1 clones were found to have the lowest calculated Levenshtein value while more complicated clones had a progressively higher value. The exception to this ~~is~~ are type-4 clones which are slightly lower than type-3 clones. This is likely due to the relatively small sample size for type-4 clones.

These average distance scores are important for several reasons. First, the average similarity score for clones had a value of 15.43, while non-clones were found to have a much higher average value of 58.4. These results are displayed in Table V. This wide gap represents the effectiveness of the calculated Levenshtein distance differentiating between clones and non-clones. A smaller variation would likely lead to many more errors in the clone detection process.

Clone Type	Avg. Levensthein Distance
T1	5.18
T2	14.53
T3	23.87
T4	18.15
Total	15.43
Nonclones	58.4

TABLE V
AVG. LEVENSTHTEIN DISTANCES

VI. RELATED WORKS

There are numerous clone detection tools which utilize a variety of methods for discovering clones. Some of which include text, lexical, semantic, symbolic and behavioral based approaches [35]. Only two known works are able to reliably detect type-4 clones. MeCC discovers clones based on the ability to compare a program's abstract memory states. While this work was successful in finding type-4 clones, there are several areas for improvement. Some of which include the ability to only analyze pre-processed C programs and an excessive clone detection time which is likely caused by the exploration of an unreasonably large number of possible program paths [24]. Krawitz [27] proposed a clone discovery technique based on functional analysis. This process was shown to detect clones of all types, but was never implemented into a reasonably functional tool. Additionally, this technique

Wrong citation: 75% of all type-3 clones! Study by Tiarks et al. did not consider type-1 and 2. Furthermore, Tiarks et al. looked at statement-level clones with at least six lines of code. This study looks at function-level clones (no size threshold is given). You cannot compare these figures!

requires a substantial amount of random data which may be a difficult and time consuming process to produce.

The most prominent area that concolic analysis has been applied to is software testing. Concolic analysis has been used for dynamic test input generation, test case generation and bug detection [26], [42], [47]. Several tools exist for performing concolic analysis. Some of which include Crest [9], Java Path Finder [10], CUTE [42] and Pex [11].

That's a somewhat arbitrary excerpt of the literature

A substantial amount of recent research has examined the ramification of clones and their relationship with the software development process. Juergens *et al.* [21] performed research on how detrimental clones actually were to the software development process. This work conducted a study on the impact of clones on commercial and open source systems and found that a high number of faults were created by inconsistent changes to clones. Alternatively, Rahman *et al.* [33] found little evidence that software maintainability was adversely affected by clones, but did discover that there was less faults in cloned code as compared to non=cloned code. Gode [15] studied the evolution of type-1 clones and found that the primary determining factor for either inconsistent or consistent changes to code was largely dependent on the specific system. Type-2 clones were examined in depth in several works [44] [25] [28] [38]. Saha *et al.* [37] analyzed how type-3 clones evolved in software and found that type-3 clones are found to be changed less consistently than type-1 and type-2 clones while all clones have a similar lifespan.

VII. Concluding Remarks

A. Threats to validity

There are certain threats to the validity of our results. First of all, our results were only run on Java and C. We do not believe the results would significantly differ if concolic clone detection was run in different languages, but without verification it is impossible to tell for certain. Concolic analysis only executes the functional aspects of an application. This means that concolic analysis will not be able to detect clones in non-functional portions of the software. This technique is also limited by the concolic analysis tools available for use, and while these tools continue to improve and are robust, they are not perfect. Many times, they are unable to traverse various portions of an application or are incapable of recognizing segments of the application for technical reasons. This would inhibit the clone detection process for these portions of the application.

A significant portion of this study was based off previous research by Krawitz [27], Roy *et al.* [35] and Kim *et al.* [24]. Therefore, our results depend to ~~a~~ a certain extent on the benchmarks provided by the aforementioned prior work. There are also numerous clone detection tools that detect clones in a variety of ways. While we were able to compare concolic analysis to several other leading detection processes, it is unreasonable to attempt to compare them to all known techniques.

B. Future Work

In the future, we plan on applying the techniques described in this paper to other areas of computing research. One area we plan on examining is how concolic analysis may assist in the field of computing security. The vast majority of malicious code is only a small variation of other malware. Additionally, significant portions of malware are reused in new iterations of intrusive software [18]. We will research if concolic analysis is capable of not only identifying malicious software, but grouping it as well. A second area we will research is how type-4 clones affect software development and how problematic they actually are. While existing research has examined many of the affects that simpler clones have on the software development lifecycle [21], to our knowledge no work has been done to analyze the effect type-4 clones have on software development.

C. Conclusion

Concolic Code Clone Detection represents a new and powerful clone detection technique. Concolic analysis executes various paths of an application~~s~~. Similar application paths represents functional similarity, and thus a code clone candidate. When compared to leading existing clone detection tools, concolic analysis was able to more accurately and reliably identify all types of clones. The proposed clone detection technique is innovative because it not only represents the first known concolic based clone detection technique, but also one of only two known processes which are able to reliably detect type-4 clones.

Project Website: A complete implementation and more in depth results regarding this study may be found at the project website http://www.se.rit.edu/~dkrutz/cccd/

[9]http://code.google.com/p/crest/
[10]http://babelfish.arc.nasa.gov/trac/jpf/
[11]http://research.microsoft.com/en-us/projects/pex/

References

[1] Mecc: Memory comparision-based clone detector, 2013.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–, Washington, DC, USA, 1995. IEEE Computer Society.

[3] Gregory V. Bard. Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In *Proceedings of the fifth Australasian symposium on ACSW frontiers - Volume 68*, ACSW '07, pages 117–124, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[4] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. Finding similar failures using callstack similarity. In *Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*, SysML'08, pages 1–1, Berkeley, CA, USA, 2008. USENIX Association.

[5] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[6] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[7] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 14–, Washington, DC, USA, 2007. IEEE Computer Society.

[8] CodePro. https://developers.google.com/java-dev-tools/download-codepro. [Online; accessed 2013-06-12].

[9] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. Xiao: tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 369–378, New York, NY, USA, 2012. ACM.

[10] Florian Deissenboeck, Benjamin Hummel, and Elmar Juergens. Code clone detection in practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 499–500, New York, NY, USA, 2010. ACM.

[11] Clone Doctor. http://www.semdesigns.com/products/clone/. [Online; accessed 2013-06-12].

[12] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20(1):3:1–3:31, July 2010.

[13] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.

[15] Nils Gode. Evolution of type-1 clones. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 77–86, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Nicolas Gold, Jens Krinke, Mark Harman, and David Binkley. Issues in clone classification for dataflow languages. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 83–84, New York, NY, USA, 2010. ACM.

[17] Simon J. Greenhill. Levenshtein distances fail to identify language relationships accurately. *Comput. Linguist.*, 37(4):689–698, December 2011.

[18] Mathur Idika. A survey of malware detection techniques. 2 2007.

[19] Mihir Jain, Rachid Benmokhtar, Hervé Jégou, and Patrick Gros. Hamming embedding similarity-based image classification. In *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*, ICMR '12, pages 19:1–19:8, New York, NY, USA, 2012. ACM.

[20] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[21] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: Research articles. *J. Softw. Maint. Evol.*, 18(2):61–82, March 2006.

[23] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, February 2013.

[24] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun 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.

[25] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, September 2005.

[26] Yunho Kim, Moonzoo Kim, YoungJoo Kim, and Yoonkyu Jang. Industrial application of concolic testing approach: a case study on libexif by using crest-bv and klee. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1143–1152, Piscataway, NJ, USA, 2012. IEEE Press.

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

[28] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Jingyue Li and Michael 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.

[30] Rao Li. A space efficient algorithm for the constrained heaviest common subsequence problem. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 226–230, New York, NY, USA, 2008. ACM.

[31] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, March 2006.

[32] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on clone stability. *SIGAPP Appl. Comput. Rev.*, 12(3):20–36, September 2012.

[33] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: what is that smell? *Empirical Softw. Engg.*, 17(4-5):503–530, August 2012.

[34] Montserrat Ros and Peter Sutton. A post-compilation register reassignment technique for improving hamming distance code compression. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 97–104, New York, NY, USA, 2005. ACM.

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

[36] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, 115, 2007.

[37] Ripon K. Saha, Chanchal K. Roy, Kevin A. Schneider, and Dewayne E. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 139–148, Piscataway, NJ, USA, 2013. IEEE Press.

[38] R.K. Saha, M. Asaduzzaman, M.F. Zibran, C.K. Roy, and K.A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 87–96, 2010.

[39] Philipp Schugerl. Scalable clone detection using description logic. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 47–53, New York, NY, USA, 2011. ACM.

[40] Sandro Schulze, Sven Apel, and Christian Kästner. Code clones in feature-oriented software product lines. *SIGPLAN Not.*, 46(2):103–112, October 2010.

[41] Carolyn B. Seaman. Software maintenance: Concepts and practice. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(2):143–147, 2001.

[42] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[43] Ruchi Shukla and Arun Kumar Misra. Estimating software maintenance effort: a neural network approach. In *Proceedings of the 1st India software engineering conference*, ISEC '08, pages 107–112, New York, NY, USA, 2008. ACM.

[44] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Softw. Engg.*, 15(1):1–34, February 2010.

[45] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Control*, 19(2):295–331, June 2011.

[46] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 67–, Washington, DC, USA, 2002. IEEE Computer Society.

[47] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 249–260, New York, NY, USA, 2008. ACM.

[48] Shuai Xie, Foutse Khomh, and Ying Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 149–158, Piscataway, NJ, USA, 2013. IEEE Press.

[49] Yang Yuan and Yao Guo. Cmcd: Count matrix based code clone detection. In *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, APSEC '11, pages 250–257, Washington, DC, USA, 2011. IEEE Computer Society.