# Examining the Effectiveness of Using Concolic Analysis to Detect Code Clones

Daniel E. Krutz and
Samuel A. Malachowsky
Software Engineering Department
Rochester Institute of Technology
{dxkvse, samvse}@rit.edu

Emad Shihab
Computer Science & Software Engineering
Concordia Unviersity
eshihab@cse.concordia.ca

## ABSTRACT

During the initial construction and subsequent maintenance of an application, duplication of functionality is common, whether intentional or otherwise. This replicated functionality, known as a code clone, has a diverse set of causes and can have moderate to severe adverse effects on a software project in a variety of ways. A code clone is defined as multiple code fragments that produce similar results when provided the same input. While there is an array of powerful clone detection tools, most suffer from a variety of drawbacks including, most importantly, the inability to accurately and reliably detect the more difficult clone types.

This paper presents a new technique for detecting code clones based on concolic analysis, which uses a mixture of concrete and symbolic values to traverse a large and diverse portion of the source code. By performing concolic analysis on the targeted source code and then examining the holistic output for similarities, code clone candidates can be consistently identified. We found that concolic analysis was able to accurately and reliably discover all four types of code clones with an average precision of .8, recall of .91, F-score of .85 and an accuracy of .99.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance;

## Keywords

Code Clones, Concolic Analysis, Software Engineering

## 1. INTRODUCTION

Software must continually change in order to keep up with user requirements, enhance its functionality, fix bugs, and repair security vulnerabilities. Prior work has shown that these code changes often result in cloned code for a variety of reasons. In many instances, developers knowingly duplicate functionality across the software system because of laziness or an unwillingness to refactor and retest the modified portion of the application. Many developers choose to avoid code clones but may not be aware that identical functionality exists in their system, on occasion unintentionally injecting clones into their application [6]. Whatever the reason, clones continue to be extremely widespread in software development; estimates have shown that clones typically amount to up to 30% of an application's source code [2, 12].

Many previous works have stated that code clones are undesirable because they often lead to more bugs and make their remediation process more difficult and expensive [2, 6]. Clones may also substantially raise the maintenance costs associated with an application [8], the importance of which is highlighted by the fact that the maintenance phase of a software project has been found to typically comprise at least 50% of the cost of a software project [23]. Inconsistent bug fixes to cloned code across a software system also increases the likeliness of further system faults [5]. Code clones, however, are not always viewed as being detrimental and may be intentionally created via certain design patterns, APIs, or organizational coding standards [21].

We define the four types of code clones using the definitions from Roy et al. [22]. Type-1 clones are the simplest, representing identical code except for variations in whitespace, comments, and layout. Type-2 clones have variations in identifiers, types, whitespace, literals, layout, and comments, but are otherwise syntactically identical. Type-3 clones are fragments which are copied and have modifications such as added or removed statements, variations in literals, identifiers, whitespace, layout and comments. Type-4 clones, the most difficult to detect, are code segments that perform the same computation, but have been implemented using different syntactic variants.

In assisting software practitioners, clone detection tools have been indispensable in detecting and managing clone-related bugs and even security vulnerabilities in software systems [4]. Of the numerous clone detection tools, most have only been able to detect the simpler clones: type-1, type-2, and type-3. To the best of our knowledge, only a few techniques are able to detect type-4 clones, the most complicated of the four [22].

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). Traditionally used in software testing to find application faults [10], concolic analysis forms the basis of a powerful clone detection tool since it only considers the functionality of the source code and not its syntactic properties. Because of this, elements that are challenging for many existing clone detection systems, such as comments and naming conventions, do not affect concolic analysis and its detection of clones. This research is important because of the ability of the technique to effectively discover all four types of code clones; few existing clone detection techniques are known to be able to do so.

*SAC '15* April 13 - 17 2015, Salamanca, Spain
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3196-8/15/04 ...$15.00.
http://dx.doi.org/10.1145/2695664.2695929.

Our study will answer the following research questions:

**RQ1:** *What types of clones is concolic analysis effective at detecting?*
We find concolic analysis is able to detect all four types of clones in both a small environment and a larger clone oracle.

**RQ2:** *How effective is concolic analysis for code clone detection?*
We measured the precision, recall, accuracy, and F-score of concolic analysis for code clone detection against two small existing oracles created by Krawitz [13] and Roy et al. [22] and a larger one built by Krutz and Le [15]. We found that concolic analysis was able to discover clones with an average precision of .8, recall of .91, F-score of .85, and an accuracy of .99. We also found that concolic analysis for clone detection compares favorably against a robust clone detection tool, MeCC [11].

In the rest of the paper, we describe how concolic analysis finds code clones, explain the types of clones it is capable of finding, and compare this technique against a leading clone detection tool.

## 2. HOW CONCOLIC CLONE DETECTION WORKS

Concolic code clone detection consists of two primary phases; an overview of this approach is shown in Figure 1. 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], CATG[2], or Java Path Finder (JPF)[3], which was used in our example. A sample segment of concolic output is shown in Table 2, and further examples are available on the project website[4]. The generated concolic output represents all executable paths that the software may take, and is broken into several *path conditions*. These conditions, which are specific to code segments, 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 must be *true*, the contingency of the path condition would be that the variable be *true*. Otherwise, this path will not be traversed [24].

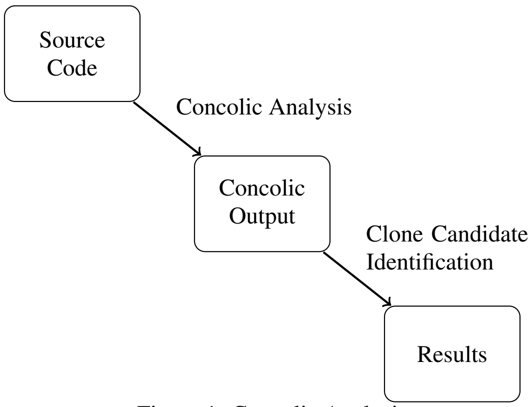

Figure 1: Concolic Analysis

Table 1 shows two type-4 clones from Roy et al. [22]. These are type-4 clones because code segment #1 uses a *for* statement and segment #2 uses a *while* statement for looping.

[1]http://code.google.com/p/crest
[2]https://github.com/ksen007/janala2
[3]http://babelfish.arc.nasa.gov/trac/jpf
[4]http://www.se.rit.edu/~dkrutz/CCCD/

Table 1: Example Type-4 Clone from Roy [22]

| **Code Segment #1** | **Code Segment #2** |
|---|---|
| void sumProd(int n){ | void sumProd(int n){ |
| float prod=1.0; | float sum=0.0; //C1 |
| float sum=0.0; //C1 | float prod=1.0; |
| for(int i=1;i<n;i++) | int i=0; |
| { | while(i<n) |
| sum=sum + i; | { |
| prod = prod * i; | sum=sum + i; |
| foo(sum, prod); | prod = prod * i; |
| }} | foo(sum, prod); |
| | i++ ; |
| | }} |

Due to space limitations, only a portion of the concolic output from running JPF on these clones is shown in Table 2. In this example, constant variable types are represented generically by "CONST" while the variable type integer is represented by a generic tag "SYMINT." Though not present in this example, other variable types are represented in a similar fashion in concolic output. Actual variable names do not appear anywhere in the output and are irrelevant to the proposed clone detection process. Concolic analysis explores the possible paths that an application can take, with similar execution paths signifying analogous functionality and is thus are indicative of a code clone candidate. Clones in *dead code* or code that is unreachable via execution paths are not be analyzed, and therefore are not discoverable via concolic analysis.

Table 2: Diff of Type-4 Clone Concolic Output

| **Concolic Segment #1** | **Concolic Segment #2** |
|---|---|
| ### PCs: 1 1 0 | ### PCs: 1 1 0 |
| original pc # = 1 | original pc # = 1 |
| CONST_1<=a_1_SYMINT | CONST_0<=a_1_SYMINT |
| SPC#0= | SPC#=0 |
| originalPC # = 1 | originalPC # = 1 |
| CONST_1<=a_1_SYMINT | CONST_0<=a_1_SYMINT |
| SPC # = 0 | SPC # = 0 |
| concolicPC # = 0 | concolicPC # = 0 |
| SPC # = 0 | SPC # = 0 |
| simplePC # = 1 | simplePC # = 1 |
| CONST_1<=a_1_SYMINT | CONST_0<=a_1_SYMINT |
| SPC # = 0 | SPC # = 0 |
| solving: PC # = 1 | solving: PC # = 1 |
| CONST_1<=a_1_SYMINT | CONST_0<=a_1_SYMINT |
| SPC # = 0 | SPC # = 0 |
| —> # = 1 | —> # = 1 |
| CONST_1<=a_1_SYMINT | CONST_0<=a_1_SYMINT |
| SPC # = 0 —> true | SPC # = 0 —> true |
| ### PCs: 2 2 0 | ### PCs: 2 2 0 |

In the concolic output in Table 2, the only differences in these compared segments are the counter values used with the "CONST" variable types used in each portion of concolic output. These differences are highlighted in the example.

The concolic output is created at the method level, and is compared to all other methods in a round-robin fashion using the Levenshtein distance measurement (the minimal number of characters that would need to be replaced to convert one string to another). As an example, if the strings "ABCD" and "BCDE" are measured, the Levenshtein distance would be 2, because "A" would need to be removed and "E" inserted into the first string to make them identical. This technique was selected for several reasons, including the impracticality of other string similarity measurement techniques. The Hamming technique, for example, may only be used with strings which are the same length [7, 20], and concolic output of even two very similar methods rarely yields output of identical length. Another example, the longest common subsequence technique, does not account for the substitution of values, only the addition and deletion of characters [18].

Because of the relative flexibility of the Levenshtein distance metric, it has proven to be especially well suited for our proposed technique. This is due in part to its ability to work with strings of different lengths and its restriction of upper and lower bounds in the calculated distances. Our distance measurement is achieved using the equation $ALV = (LD/LSL) \times 100$. The Average Levenshtein Value (ALV) is computed by dividing the Levenshtein Distance between two files (LD) by the Longest String Length (LSL) of the two strings being compared and then multiplying by 100. While only a portion of the concolic output is shown in Table 2, the Levenshtein distance between the two complete sets of output was 25, and the longest string length was 2,216. This means that our formula to calculate the Levenshtein distance between this output is $ALV = (25/2216) \times 100 = 1.13$, which indicates a strong similarity score, and thus a strong likelihood of a code clone.

We use a Levenshtein threshold score of 30 in our analysis to determine if two compared items are code clone candidates. Our first step in determining this as the most appropriate Levenshtein value was to produce concolic output from the oracles by Krawitz [13], Roy et al. [22] and Krutz and Le [15] using Levenshtein scores of 0-40 with 10 point increments as a basis for determining clones. To obtain the optimal number, we compared the precision, recall, F-score, and accuracy scores of each increment and found that for all of the code bases, the Levenshtein value of 30 produced the best rates.

We combined the precision, recall, F-score, and accuracy values of the code bases and placed them into two charts to better visualize the effects of using the different Levenshtein scores to determine clones. Figure 2 displays the results of various Levenshtein values in discovering clones in a single class as defined by Krawitz and Roy et al. Figure 3 shows a similar analysis the results from the Krutz and Le [15] oracle.

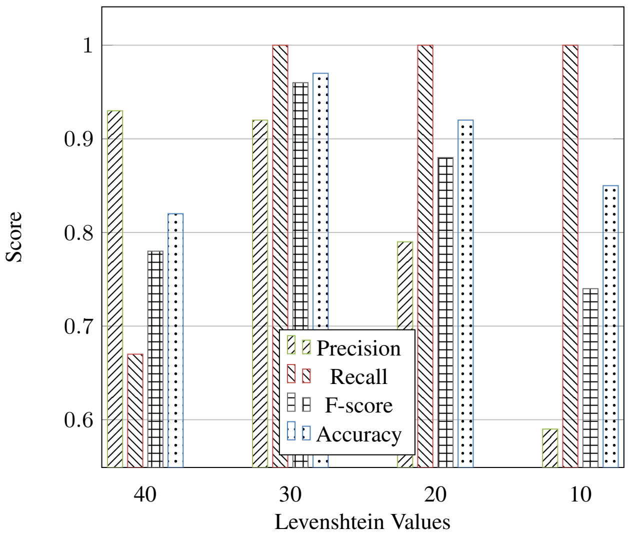

Figure 2: Levenshtein Impact In Single Class

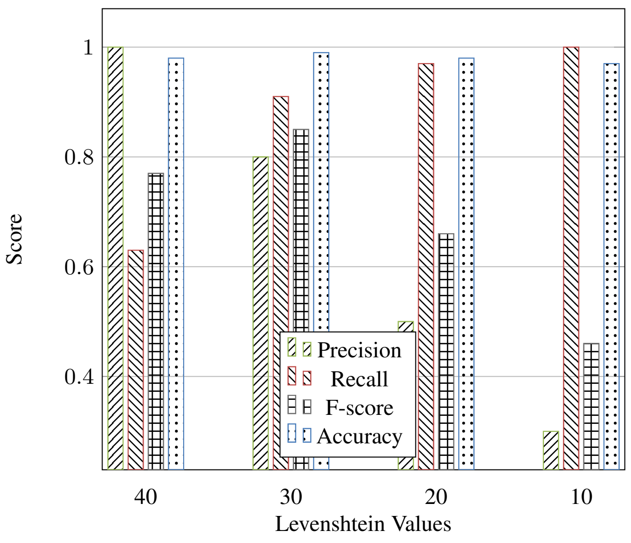

Figure 3: Levenshtein Impact On Oracle by Krutz and Le [15]

A higher Levenshtein threshold score is likely to aid in the discovery of more clones, but will also lead to more false positives, creating lower precision but higher recall. Conversely, a lower Levenshtein threshold score will find fewer actual clones, but also have less false positives leading to low recall, but higher precision. This is because a higher Levenshtein score means that the similarity threshold for noting cloned items will be reduced. Different Levenshtein values may be selected depending on their desired levels of precision, recall, F-score, and accuracy.

## 3. EVALUATION

In the following sections, concolic analysis for clone detection will be evaluated against two small oracles created by Krawitz [13] and Roy et al. [22] and a larger oracle built by Krutz and Le [15] to determine what types of clones concolic analysis is capable of finding, along with its accuracy, precision, recall, and F-score.

### 3.1 Types of Clones Discovered

The C-based applications were analyzed via concolic analysis using the Concolic Code Clone Detection (CCCD) tool [14, 16]. In the previous paper, we demonstrated the ability of concolic analysis to effectively discover all types of code clones in a very small environment but did not thoroughly analyze the technique. We will build on these results and further evaluate concolic analysis for clone detection.

Table 3: Concolic Analysis Finding Clones on Single Class

| Language | T1 | T2 | T3 | T4 | Total |
|---|---|---|---|---|---|
| **Java** | 5 | 6 | 6 | 6 | 23 (96%) |
| **C** | 5 | 6 | 7 | 4 | 22 (92%) |
| **Total Possible** | 5 | 6 | 7 | 6 | 24 |

**RQ1:** ***What types of clones is concolic analysis effective at detecting?*** The initial step of evaluating concolic analysis for code clone detection was to evaluate it against four clones defined by Krawitz, and 16 by Roy et al. These 20 defined clones were added to a Java and C file. The results in Table 3 indicate the ability of concolic analysis to find a wide range of clones in this small, controlled environment in both C and Java applications. We used

CCCD for the C code, and developed a small prototype based upon JPF for the Java code.

Within the limited Java implementation, the concolic analysis-based technique was able to detect 96% of all clones. The only clone which concolic analysis was unable to detect was a type-3 clone as defined by Roy et al., as JPF was unable to traverse all paths of this method for technical reasons including its inability to perform analysis on several unsupported variable types (float, byte, and short). This limitation ultimately affects the concolic analysis clone identification process specifically when applied to Java.

A similar C file containing the clones of Krawitz and Roy et al. was then examined for clones. Concolic analysis was able to detect 92% of all clones; the only clones it 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 similar 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. Our technique did, however, find all other instances of type-4 clones. The size of the examined functions did not have a significant impact on the ability of any of the examined processes in detecting clones.

This small example demonstrates that concolic analysis for code clone detection is capable of finding all four types of clones in both Java and C.

## 3.2 The Effectiveness of Concolic Analysis For Code Clone Detection

Our next step was to evaluate the effectiveness of code clone detection in terms of precision, recall, F-score, and accuracy.

**RQ2:** ***How effective is concolic analysis for code clone detection?***
In order to evaluate the effectiveness of concolic analysis for code clone detection, we used a function level clone oracle created by Krutz and Le [15] since it is the only known oracle to contain all four types of code clones explicitly defined. We ran concolic analysis for code clone detection and measured the accuracy, precision, recall, and F-score of this technique against this clone oracle.

The oracle was created by first randomly selecting 3-6 classes from Apache, Python, and PostgreSQL. A specially made comparison tool allowed several researchers to independently and manually compare all functions and record if the compared functions were code clones, and, if so, what type of clone they were. Several leading clone detection tools were then run against the code base with their findings being recorded. These tool results were then used by the researchers to identify any clones which they may have missed for further analysis. The ultimate decision of whether or not two compared functions represented a clone fell upon the researchers and not any tool. When researchers disagreed if two compared functions represented or on the type of clone, a discussion took place until a consensus could be reached. While CCCD was one of the selected tools used as input for this oracle, all clone decisions were manually verified and tools were never the deciding factor as to what constituted a clone. We do not feel like this negatively impacted the results.

Precision, recall, F-score, and accuracy are important factors in evaluating clone detection tools [29]. They should not return too high of a rate of false positives, but also not miss a significant portion of code clones. The definitions we used for precision, recall, F-score, and accuracy (which will fall between 0 and 1) are described below:

1. **Precision:** Ratio of the clone pair which a tool reports that are true clones, not false positives.
2. **Recall:** Ratio of the clone pairs in a system that a tool is able to detect.
3. **F-score:** Considers precision *and* recall to measure the accuracy of a system. It is calculated as $2 \times (\frac{precision \times recall}{precision + recall})$. Sometimes referred as F1 or F-measure.
4. **Accuracy:** Percentage of elements classified correctly.

In order to evaluate the effectiveness of concolic analysis for clone detection against an existing technique, we compared CCCD against MeCC, a tool which is capable of discovering all four types of code clones [11]. We ran MeCC against the Krutz and Le [15] oracle using a variety of values for its two input parameters, similarity, and minimum entry size. We used the settings which produced highest rates of precision, recall, accuracy, and F-score values against the clone oracle which were a similarity of 80 and a minimum line entry of 4. We then ran CCCD against the same oracle. The resulting averages for each tool are shown in Table 4.

Table 4: Average Precision, Recall, F-Score & Accuracy

| Tool | Precision | Recall | F-Score | Accuracy |
|---|---|---|---|---|
| **MeCC** | .6 | .47 | .46 | .96 |
| **CCCD** | .8 | .91 | .85 | .99 |

These results demonstrate the effectiveness of concolic analysis for clone detection against a leading tool. While both techniques are able to achieve a high rate of accuracy, CCCD has a much higher F-score and recall than MeCC.

Concolic analysis has been shown to be a powerful clone detection method which is not only able to discover a wide range of clone types (including type-4), but is also able to find them with a high rate of precision, recall, F-score, and accuracy.

## 4. RELATED WORKS

There are numerous clone detection tools which utilize a variety of methods for discovering clones including text, lexical, semantic, symbolic, and behavioral based approaches [11, 22]. However, only a few are known to be 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 such as its limitation in analyzing pre-processed C programs and an excessive clone detection time, likely caused by the exploration of an unreasonably large number of possible program paths [11]. Krawitz [13] proposed a clone discovery technique based on functional analysis which was shown to detect clones of all types, but was never implemented into a reasonably functional tool. This technique's analysis also requires a substantial amount of random data for determining boundary values, which may be a difficult and time consuming process. There are numerous other clone detection tools which have been used in previous research. Roy et al. [22] carried out a thorough analysis of many tools in 2009 which describes many of the different types of clone detection tools and techniques. Subsequent works have compared tools, but on a smaller scale [1, 25].

The most prominent area that concolic analysis has been applied to thus far is software testing, specifically for dynamic test input generation, test case generation, and bug detection [24, 28]. Several tools exist for performing concolic analysis, including Crest, Java Path Finder, CUTE [24], and Pex[5].

[5]http://research.microsoft.com/en-us/projects/pex

We chose to use the code clone oracle created by Krutz and Le [15] in 2014 since it explicitly contains all four types of code clones, but there are several other prominently used clone oracles which have been proposed in previous research. Tempero [26] described a collection of 1.3M method-level-clone-pairs from 109 different systems. The goal of this work was to create a similar data set for clone research. While this work was profound, much of the data has a low level of confidence and requires further work and analysis.

Lavoie and Merlo [17] created a clone oracle set containing type-3 clones using the Levenshtein metric. There was no mention of type-4 clones in this oracle. Bellon et al. [3] created a robust clone oracle which has been used in a substantial amount of research. This work was recently extended upon by Murakami et al. [19]. Unfortunately, neither of these oracles contain any explicitly defined type-4 clones.

## 5. THREATS TO VALIDITY

There are several threats to the validity of our results. First, our results were only run on Java and C. We do not believe that they 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, meaning that it 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. In some cases they are unable to traverse various portions of an application or are incapable of recognizing segments of the software for technical reasons. This inhibits the clone detection process for these portions of the application. Finally, the followed path conditions depend upon the control flow graph and its predicates, meaning that concolic analysis for clone detection is still dependent upon its implementation. While it is less dependent than syntax or token based clone detectors, many code instances of identical semantics or different implementations will not be detected by concolic analysis for clone detection. Concolic analysis may also be a slow and resource intensive process which could adversely affect an implementation of our technique on a very large code-base.

A significant portion of this study was based off previous research by Krawitz [13], Roy et al. [22], Krutz and Le [15], and Kim et al. [11]. Our results, therefore, depend to a certain extent on the benchmarks provided by the aforementioned prior work. Manually finding type-4 clones in source code is extremely difficult and there are only few automated techniques which are known to reliably find type-4 clones. This makes it very difficult to test a new mechanism in finding these clones since there are very few benchmarks to be evaluated against.

The classification of clones and their type is a difficult and imprecise task [27], so many researchers will likely disagree with the classification of clones from our oracles. This is a problem which is not at all unique to our work and affects other research as well [17]. While many works recognize type-4 clones [3, 22] other recent research does not acknowledge their existence [6, 17], so there is some fragmentation in the code clone community as to whether type-4 clones even exist. This could mean that there are other tools which are capable of finding type-4 clones, but simply made no effort to do so. In spite of these possible limitations, we are confident that concolic analysis is able to discover type-4 clones as is exemplified by our evaluation using our chosen oracles.

We compared concolic analysis (using CCCD) against MeCC and evaluated their rates of precision, recall, F-score, and accuracy. One potential problem with this comparison is that MeCC finds clones at the sub-method level, while CCCD is only capable of discovering clones at the method level. In order to mitigate this problem, we analyzed all clones identified by MeCC which were at the method level and used them for our comparison. This means that MeCC will likely discover more clones than CCCD , but could also lead to more false positives. While we acknowledge that this could create a bias of our findings, the purpose of this comparison was not to state that CCCD was necessarily better than MeCC, but to merely demonstrate that concolic analysis is a powerful mechanism for clone detection.

## 6. FUTURE WORK

While we demonstrated that concolic analysis is capable of reliably and accurately discovering all four types of clones, future work may be conducted in several key areas. We only compared concolic analysis for clone detection against MeCC. Future work should be done to evaluate our technique against other leading clone detection tools such as SeByte [9], CCFinderX[6], ConQat[7], ctCompare[8], Deckard[9], iClones[10], Simian[11], Simcad[12], and Nicad[13].

We used an oracle created by Krutz and Le [15] since it contained all four types of clones. We did not choose to use oracles from Bellon et al. [3] or Murakami et al. [19] since they did not explicitly contain type-4 clones. However, future work may be done to expand on our findings using these oracles.

Finally, we have demonstrated that concolic analysis for clone detection may only discover clones at the method level. Future work can be done to implement a concolic-based solution which may locate clones at a more granular level. The biggest obstacle in creating this solution is with the comparison process that our technique uses for discovering clones. Currently, the concolic output from each method is compared to the others in a round robin fashion. Comparing snippets of code with one another would be a virtually insurmountable task. Future work may be done to develop a more efficient comparison process.

## 7. CONCLUSION

Concolic code clone detection represents a new and powerful clone detection technique which we have demonstrated to be capable of finding all four types of code clones with high precision, recall, accuracy, and F-score values. We evaluated concolic analysis for clone detection using a small C and Java based oracle, and then with a larger oracle comprised of C code. The proposed clone detection technique is innovative because it not only represents the first known concolic-based clone detection technique, but is also one of only a few known processes which is able to reliably detect type-4 clones.

## References

[1] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti. Software clone detection and refactoring. *International Scholarly Research Notices*, 2013, 2013.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings*

[6]http://www.ccfinder.net/ccfinderx.html
[7]https://www.conqat.org/
[8]http://minnie.tuhs.org/Programs/Ctcompare/
[9]https://github.com/skyhover/Deckard
[10]http://www.softwareclones.org/iclones.php
[11]http://www.harukizaemon.com/simian/
[12]http://homepage.usask.ca/~mdu535/tools.html
[13]http://www.cs.usask.ca/~croy

*of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, 2007.

[4] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. 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.

[5] F. Deissenboeck, B. Hummel, and E. 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.

[6] E. Duala-Ekoko and M. 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.

[7] M. Jain, R. Benmokhtar, H. Jégou, and P. 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.

[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. 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.

[9] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 36–42, Piscataway, NJ, USA, 2012. IEEE Press.

[10] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. 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, Feb. 2013.

[11] 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.

[12] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, Sept. 2005.

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

[14] D. E. Krutz. *Concolic Code Clone Detection*. PhD thesis, Nova Southeastern University, 2012.

[15] D. E. Krutz and W. Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA, 2014. ACM.

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

[17] 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.

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

[19] H. Murakami, Y. Higo, and S. Kusumoto. A dataset of clone references with gaps. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 412–415, New York, NY, USA, 2014. ACM.

[20] M. Ros and P. 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.

[21] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 18–33, Feb 2014.

[22] 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.

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

[24] K. Sen, D. Marinov, and G. 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.

[25] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Software Clones (IWSC), 2013 7th International Workshop on*, pages 16–22. IEEE, 2013.

[26] E. Tempero. Towards a curated collection of code clones. In *Proc. IWSC*, pages 53–59. IEEE, 2013.

[27] 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.

[28] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. 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.

[29] M. F. Zibran and C. K. Roy. Ide-based real-time focused search for near-miss clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1235–1242, New York, NY, USA, 2012. ACM.