Noname manuscript No.
(will be inserted by the editor)

Examining the Effectiveness of Using Concolic Analysis to Detect Code Clones

Daniel E. Krutz · Emad Shihab · Samuel A. Malachowsky

the date of receipt and acceptance should be inserted later

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 simply defined as multiple code fragments that produce similar results when provided the same input. While there are an array of powerful clone detection tools, most suffer from a variety of drawbacks including the inability to accurately and reliably detect all four types of clones.

This paper presents a new method 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. In order to measure the effectiveness of the technique, we performed a case study and found that concolic analysis was able to detect 92% of known clones in a controlled environment, including a significant number of harder-to-find type-4 clones. Concolic analysis was also able to consistently and effectively locate existing clones in several open source applications with an average precision of 83% and recall of 93%, both of which were significantly higher than existing clone detection tools to which they were compared to.

[rewrite abstract based on findings] [Clean up format based upon new publication]

Keywords Code Clones · Concolic Analysis · Software Engineering

D. Krutz
Department of Software Engineering, Rochester Institute of Technology, NY USA
E-mail: dxkvse@rit.edu

E. Shihab
E-mail: emad.shihab@rit.edu

S. Malachowsky
E-mail: samvse@rit.edu

1 Introduction

Software must continually change in order to keep up with user requirements, enhance its functionality, fix bugs, or repair security vulnerabilities. Prior work has shown that these code changes often results 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. Careful developers, who know to avoid code clones, may not be aware that identical functionality exists in the system, unintentionally injecting clones into the application [5, 13, 24, 35]. Whatever the reason, clones continue to be extremely widespread in software development; estimates have shown that clones typically amount to between 5% and 30% of an application's source code [7, 28, 43].

Many previous works have stated that code clones are undesirable since they often lead to more bugs and make their remediation process more difficult and expensive [5, 7, 13, 36]. Clones may also substantially raise the maintenance costs associated with an application [21], the importance of which is highlighted by the fact that the maintenance phase of a project has been found to encompass between 40% and 90% of the totsl cost of a software project [9, 14, 15, 44, 46, 49]. Ultimately, unintentionally making inconsistently applied bug fixes to cloned code across a software system increases the likeliness of further system faults [12].

There are four types of code clones generally recognized by the research community. Type-1 clones are the simplest, representing identical code except for variations in whitespace, comments, and layout [8]. Type-2 clones are syntactically similar except for variations in identifiers and types. Type-3 clones are two code segments which are syntactically different due to altered or removed statements. Type-4 clones, the most difficult to detect, are two code segments which have considerable differences syntactically, but produce identical results when executed [11, 16].

To assist software practitioners in detecting and managing code clones, clone detection tools have been indispensable in detecting clone-related bugs and even security vulnerabilities in software systems [11]. Of the numerous clone detection tools, most have only been able to detect the simpler clones: type-1, type-2, and type-3. Type-4 clones, the most difficult to detect [39, 52], have, to the best of our knowledge, only two processes able to reliably detect them. MeCC, capable of reliably detecting type-4 clones, suffers from several drawbacks, including the ability to only analyze pre-processed C programs [27].

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 [26, 29], 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. Because of this, elements that are challenging for existing clone detection systems such as comments and naming conventions do not affect concolic analysis and its detection of clones.

This research is innovative because, to our knowledge, no previous attempts have been made in using concolic analysis in clone discovery. Any technique which can

effectively discover all four types of code clones is important, since, at the present time, so few clone detection processes are able to do so.

Concolic Code Clone Detection (CCCD) is a fully functional tool that uses concolic analysis for clone detection. Concolic analysis is performed on the target application using CREST [1]. First, a Java component uses CTAGS [2] 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. Finally, a report displays the detected code clone candidates. This tool, installation instructions, and further details may be found on the project website [1] and was published in a previous work [32].

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 types of clones, in both, a controlled environment and in several open source applications. In a controlled environment using clones identified by previous research, concolic analysis was able to detect 100% of type-1, type-2, and type-3 clones, and 67% of type-4 clones. **[update values]**

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 find that concolic analysis compares very favorably to these tools. Using manually identified clones in several open source systems, concolic analysis consistently discovered clones with a higher rate of precision, recall and F-score when compared to several leading existing clone detection tools. Concolic analysis averaged 83% for precision, 93% for recall and an 88% F-Score while the next leading tool, Nicad, was only able to achieve scores of 66%, 80%, and 59%, respectively. **[update values]**

The remainder of the paper is organized as follows. Section 2 describes how concolic analysis may be used to to detect software clones. Section 3 evaluates the ability of concolic analysis in identifying clones in relation to existing tools. Section 4 conveys interesting results from the research. Section 5 discusses related works in clone detection and concolic analysis. Section 6 details some threats to the findings of this work. Section 7 provides concluding remarks and future research directions for this work. **[update section]**

2 How Concolic Clone Detection Works

In explaining how concolic code clone detection works, a breakdown of the concept and illustration of it in action is needed. First, we will describe how concolic analysis is performed on two cloned methods. This will include the use of the Levenshtein distance algorithm in measuring the similarity of two sets of concolic output. Next, we will provide a motivating example using a cloned method which has been analyzed by several leading clone detection tools. Finally, we will briefly explain some of the

[1] https://github.com/jburnim/crest/
[2] http://ctags.sourceforge.net

Code Segment #1	Code Segment #2

`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);`
`  }`
`}`
	`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);`
	`  }`
	`}`

Table 1: An Example of Type-2 clones from Roy

shortcomings of these tools and why concolic analysis was successful in finding the clones.

2.1 Concolic Clone Detection Technique

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 Table 1. These are derived from clones presented by Roy *et al.* [42].

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 [3], Java Path Finder (JPF) [4], or CATG [5].

An abbreviated example segment of concolic output is shown in Listing 1; the complete output is may be viewed on the project website [1]. 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 [45].

[3] http://code.google.com/p/crest/

[4] http://babelfish.arc.nasa.gov/trac/jpf/

[5] https://github.com/ksen007/janala2

Listing 1: Example Concolic Output

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

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

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

In Listing 1, constant variable types are represented generically by "CONST" while the variable type integer is represented by a generic tag "SYMINT." Though not present above, other variable types are represented in a similar fashion in concolic output such as this. Actual variable names do not appear anywhere in the output and are irrelevant to the concolic analysis technique. When comparing the output from the type-2 clones in Table 1, one of the primary differences would be that *sum* would be defined as a *double* variable type in 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 noted to be code clone candidates. 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 excecution paths will not be analyzed by concolic analysis, and therefore are not discoverable via concolic analysis. To demonstrate this functionality, clones as defined by Roy *et al.* [42] in Table 1 were analyzed using concolic analysis. A portion of the generated concolic output, demonstrated in Table 2, was then compared for variations. The only difference between the two sets of concolic output is the method name displayed in the first line of each file (an abbreviated listing is shown; full results are available on the project website).

In order to measure the similarity between sets of the concolic output, the Levenshtein distance measurement is used. This is the minimal number of characters that would need to be replaced to convert one string to another [6, 17]. 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, but the main motivation related to the impracticality of other string similarity techniques in clone detection using concolic analysis. The Hamming technique, for example, may only be used with strings which are the same length [19, 38], and concolic output of even two very similar methods rarely yields output of identical length. Another possibility, the longest common subsequence technique, does not account for the substitution of values, only the addition and deletion of characters [34], which also proves problematic.

Because of the relative flexibility of the Levenshtein distance metric, it has proven to be especially well suited for this particular task in clone detection. Due in part to

Concolic Segment #1	Concolic Segment #2
sumProd1(a);	sumProd2(a)
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

Table 2: Diff of Concolic Output

its ability to work with strings of different lengths and its restriction of upper and lower bounds in the calculated distances, normalization is achieved via Equation 1 below. 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.

$$ALV = (LD/LSL) \times 100 \tag{1}$$

In order to evaluate the effectiveness of concolic analysis in detecting clones, we created two prototypes. The first was able to analyze source code written in C and utilized CREST in generating the necessary concolic output. The second was created using JPF and examines source code written in Java, though it was not as robust. This Java-based prototype was important because it demonstrated several key concepts, including the assertion that this technique of detecting clones is language agnostic, and that other concolic analysis tools may be used in support of this process. An overview of the entire clone detection technique is shown in Figure 1.

2.2 Preliminary Experiments

A simple comparison was conducted against several leading clone detection tools to further evaluate concolic analysis for code clone detection. This was accomplished using a type-3 clone as defined by Roy *et al.* [42] and is shown in Table 3. The tools used for our analysis are described below.

[add more tools] [only compare tools that work at the method level?]*[Dan says: It seems like we are often compare tools at different levels, and this makes things apples to oranges]*

Simian: A text-based tool which uses source transformations and regular expressions to find clones. Simian supports a wide range of languages including Java, C#, C, C++, ASP, and Ruby. Previous work has demonstrated that this tool finds

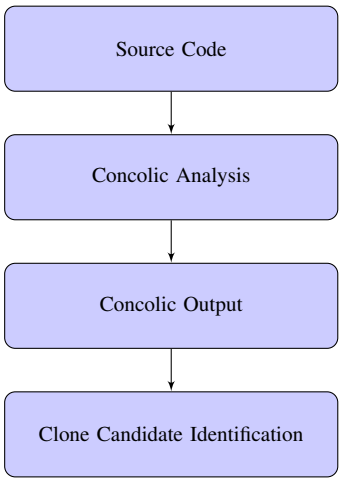

Fig. 1: Concolic Analysis

type-1 and type-2 clones reasonably well, but struggles at discovering the more complicated type-3 and type-4 clones [3, 42].

Nicad: A text-based hybrid tool that combines the advantages of text and tree-based structural analysis. Clone identification and normalization are conducted using pretty printing and longest common subsequences. Nicad is compatible with C, Java, C#, and Python and has been found to have the ability to detect type-1, type-2, and type-3 clones, but is known to struggle with type-4 clones [40, 41, 42].

CloneDR: A tree-based clone detection tool which uses hashing and dynamic programming. Annotated parse trees are created using a compiler generator. These trees are then hashed into buckets and compared with one another in the search for clones. This tool is available for numerous languages including C, C++, C#, Java, and Python and has been found to detect type-1 and type-2 clones very well, type-3 clones with limited effectiveness, and significantly struggles with the identification of type-4 clones [2, 7, 37, 42].

MeCC: Discovers clones by comparing the abstract memory states of an application. A path-sensitive semantic-based path analyzer is first run to estimate the memory states of each method's exit point, then these memory states are then compared to discover clones. Currently only supports preprocessed C programs. The authors of this tool have stated that this tool is able to find all four types of clones with a high rate of accuracy, but there has been a limited amount of other work evaluating this tool [4, 25, 27].

Simcad: A powerful, recently developed clone detection tool derived from a previously created tool called *simhash* designed to quickly discover exact and near miss clones in large applications. Simcad discovers clones at either the block or function level and is compatible with C, Java, and Python applications. Clone candidates are discovered through the use of three primary phases: pre-processing, detection, and output generation [48].

Code Segment #1

```
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);
    }
}
```

Code Segment #2

```
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);
  }
}
```

Table 3: An Example of Type 3 Clones from Roy *et al.*

[Add in further description of the other tools used and if they could find this clone, and if they could not]

Each of the above detection tools were used, but concolic analysis and MeCC were the only tools able to detect the type-3 clone contained within the example code. CloneDR and Simian were unable to detect this clone because they are tree and text-based clone detection tools; even though the two examined methods are functionally equivalent, they are syntactically different. This is something that these two types of clone detection techniques often struggle with. CodePro is a closed source application and makes no mention of what type of technique it uses for clone detection [20], so it is unclear why it was unable to detect this clone. Nicad, essentially a hybrid-based solution, is a text-based detection system which also relies upon the benefits of a tree-based structural analysis and code normalization in order to discover clones [42]. This reliance of the text of the source code likely hindered its ability to find the clone in this example. The size of the functions being examined does not affect the ability of any tools to detect these clones.

MeCC was able to detect this clone since the memory states between the two clones were identical, even though there are syntactic differences between the two compared methods. 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 may cause problems for text and syntax-base clone detection tools do not adversely affect the ability of concolic analysis to detect clones. Simcad was also able to identify this clone because it relies on a data clustering algorithm to discover clones, which is not based on the syntactic nature of the code being analyzed [48].

3 Evaluation

[Provide Brief introduction here] In the following sections, concolic analysis for clone detection will be evaluated individually and against other leading existing clone detection tools.

3.1 Clone Dataset

[Describe the datasets that we used]

3.2 Technique Comparisons

[Provide a better introduction to this]

Our Java-based prototype used Java Path Finder (JPF) to produce the concolic output of the target application. Unfortunately, due to technical limitations of the tool, we were unable to analyze any reasonably sized open source applications using JPF. Of the most profound was its inability perform concolic analysis on several variable types including float, byte, and short, significantly limiting the amount of methods JPF was able to analyze. Without further development, continued use of this tool would have led to inaccurate results since such a large number of methods would have had to been ignored or allowed to produce errors during concolic analysis. Because of this, the Java implementation will only examine proof of concept classes and will not analyze the more complicated Java-based open source applications or similar-sized code bases.

The C-based applications were analyzed by concolic analysis using the Concolic Code Clone Detection (CCCD) tool, published in a previous work by the authors [32]. In the previous paper, we demonstrated the ability of concolic analysis to effectively discover all types of code clones in a small, controlled environment. We will build on these results and further compare concolic analysis against several leading existing clone detection tools using clones as defined in previous research by Krawitz [30] and Roy *et al.* [42].

RQ1: ***What types of clones is concolic analysis effective at detecting?*** **[update all of this data]**

The initial step of evaluating concolic analysis for code clone detection was to evaluate it against 4 clones defined by Krawitz, and 16 by Roy *et al.*. These 20 defined clones were added to a single Java and C file, and several leading clone detection tools were selected for comparison purposes. For Java, these were CodePro, CloneDR, Simian, Simcad, and Nicad. For C-based applications, Simcad, MeCC, and Nicad were chosen. CodePro and MeCC were only capable of finding clones in Java and C-based applications, respectively. In all examples, the default or recommended settings were used, and the results are shown in Table 4. Based on the anemic results early on, Simian and CloneDR were not selected for further evaluation; the remaining tools were subjected to further analysis.

Within the limited Java implementation, the concolic analysis based technique was able to detect 96% of all clones, Simcad 83%, Nicad 58%, CodePro 46%, and CloneDR 38%. The only clone which concolic analysis was unable to detect was a type-3 clone as defined by Roy *et al.*. JPF, the implementation used by CCCD, 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).

Table 4: Comparison of tools on single class

Application	Tool	T1	T2	T3	T4	Total
Java	CloneDR	5	4	0	0	9 (38%)
	CodePro	2	3	4	2	11 (46%)
	Simian	5	0	0	0	0 (21%)
	Nicad	3	4	4	3	14 (58%)
	Simcad	5	5	6	4	20 (83%)
	CCCD	5	6	6	6	23 (96%)
C	CloneDR	5	4	0	0	9 (38%)
	Simian	3	2	2	1	8 (33%)
	MeCC	5	6	6	3	20 (83%)
	Nicad	5	4	4	3	16 (67%)
	Simcad	5	6	7	3	21 (88%)
	CCCD	5	6	7	4	22 (92%)
Total Possible		5	6	7	6	24

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, Simcad 88%, MeCC 83%, and Nicad 67%. While Simcad, Nicad, and MeCC were all able to discover at least one type-4 clone, none found as many as concolic analysis. All default settings for the clone detection tools were used with complete results being available on the project website [1]. In the controlled environment, MeCC, Nicad, and Simcad achieved the best results and were therefore selected for further analysis on larger scale real world applications.

The size of the examined functions did not have a significant impact on the ability of any of the examined processes in detecting clones. 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 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.

RQ2: ***How does concolic analysis based clone detection compare to other leading clone detection tools?*** **[update all of this]**

[Talk about and use existing oracles, Bellon etc.....]

In order to compare concolic analysis for clone detection against leading tools in existing systems, we first needed an oracle with predefined clones of all types. Since no known, substantial oracle existed with all four types of clones present, we generated our own. This dataset has been presented and further described in a previous publication [31]. To create this oracle, we first selected several open source applications-specifically Apache 2.2.14, Python 2.5.1, and PostgreSQL 8.5 primarily because they had already been used in previous clone detection research [27] and since they are all widely known, open source applications which are publicly available. These applications were selected as-is and had no source code alterations performed. Since we would need to manually verify which methods in the applications were clones, and if so, what type, we then randomly chose 3-6 classes from each application to analyze.

The only selection criteria were that the classes needed to contain at least ten methods each, allowing a reasonably sized cross-section for analysis.

Even though only a subset of each application was analyzed, every method would be compared to each other. Because of this, the number of potential clones to be analysed was exponentially large. Within the three applications, there were a total of 45,109 possible clones to verify, a number too large to inspect manually. In order to address this, we selected a statistically significant number of random clone combinations to examine with the goal of having a confidence level of 99% and a confidence interval of 5. We created an open source tool, CloneInspector [6], to automatically load the selected candidate clone comparisons for manual on-screen analysis. This tool also allowed the user to record whether the comparison was or was not a clone, and, if so, what type of clone it was. Two researchers familiar with code clones independently completed the analysis and any discrepancies with the findings were discussed until an agreement could be reached. The oracle was created before any clone detection efforts ensued to reduce bias during manual analysis. Full results are available on the project website [7].

3.3 Types of Discovered Clones

Most clone detection tools have a variety of input parameters that can be set before analysis. Typically, these include, but are not limited to, the similarity score used to determine if two compared items represent a clone, and the minimum number of required lines for two segments to be recorded as clones. As expected, more stringent settings lead to less false positives, but also less actual clones being discovered. Conversely, lowering these standards may lead to more clones being found, but an inappropriately high number of false positives. The goal is to find the most appropriate balance, optimizing clones found and false positives.

In order to determine the most appropriate settings to use for each tool, we evaluated each using a variety of size and similarity settings against our created oracle. Evaluation criteria included precision, recall, F-score, accuracy, and the number of different clone types found. A similarity score of 70 with a minimum size of 20 was found to achieve the highest scores for MeCC. Nicad has no size parameters and had the best results with a similarity rating of 50. The only relevant settings for Simcad were language, granularity (block of function), and clone type. All compared clone detection tools were made to search for clones at the function level, since this is the same level at which CCCD discovers clone candidates. Further results may be found in the appendix of the paper.

Each tool's findings (including CCCD) by total of each clone type are presented in Table 5. CCCD found the most total clones, as well the most type-2, type-3, and type-4 clones. None of the tools found a significant number of type-1 clones, which is due to the lack of type-1 clones identified in the oracle. This may be largely attributed to the fact that type-1 clones are the easiest to manually locate, and would have likely

[6] https://https://github.com/cloneoracle/CloneInspector/

[7] http://phd.gccis.rit.edu/weile/data/cloneoracle/

been recognized and removed by developers in the source code. Nicad was the only other tool to discover any type-4 clones, finding a single pair, while CCCD found 32.

Table 5: Clones Found by Type

Tool	Source Example	T1	T2	T3	T4	Total
Mecc	**Apache**	1	16	0	0	17
	P-SQL	0	2	0	0	2
	Python	0	6	6	0	12
	Total	**1**	**24**	**6**	**0**	**31**
Simcad	**Apache**	0	12	0	0	12
	P-SQL	0	0	0	0	0
	Python	0	5	7	0	12
	Total	**0**	**17**	**7**	**0**	**24**
Nicad	**Apache**	1	16	0	0	17
	P-SQL	0	2	1	1	4
	Python	0	6	8	0	14
	Total	**1**	**24**	**9**	**1**	**25**
CCCD	**Apache**	1	17	0	8	26
	P-SQL	0	15	9	24	48
	Python	0	6	4	0	10
	Total	**1**	**38**	**13**	**32**	**84**

3.4 Accuracy, Precision & Recall

In addition to their ability to find clones, precision and recall are important factors in evaluating clone detection tools [53]. The tool should not return too high of a rate of false positives, while it should also not miss a significant portion of code clones - striking an ideal balance between the two. In order to calculate accuracy, precision and recall, we used the data previously attained from running concolic analysis and existing tools against our oracle. To study the prediction accuracy, we built two multivariate logistic regression models: one that uses all of the metrics and one that uses a smaller set of statistically and minimally collinear metrics. The logistic regression models are designed predict the likelihood of a file being defect prone or otherwise. The output is given as a value between 0 and 1; we classified values above 0.5 as defect prone, with the remainder classified defect free. The classification results of the prediction models were stored in a confusion matrix, as shown in Table 6.

Table 6: Confusion matrix

		True Class	
		Yes	No
Predicted	Yes	a	b
	No	c	d

The performance of the prediction model is measured in four different ways. Values for each will range from 0 to 1, with a 1 being favorable:

1. **Precision:** Relates the number of files predicted *and* observed as defect prone to the number of files predicted as defect prone. It is calculated as $\frac{a}{a+b}$.
2. **Recall:** Relates the number of files predicted *and* observed as defect prone to the number of files that actually had defects. It is calculated as $\frac{a}{a+c}$.
3. **F-Score:** Considers precision *and* recall to measure the accuracy of a system. It is calculated as $2 \times (\frac{precision \times recall}{precision + recall})$.
4. **Accuracy:** Percentage of elements classified correctly. The highest attainable value is 1.0. It is calculated as $\frac{a+d}{a+b+c+d}$.

Table 7 displays the precision, recall, F-score and accuracy for Nicad, MeCC, Simcad, and CCCD using a Levenshtein distance of 30. Only these tools are shown based on their results in Table 5, and because many of the calculations would have been unable to return reliable values due to the tool's inability to find any clones for a target application. A complete listing of all results and tool types may be found in Table 9 in the appendix.

Table 7: Precision, Recall, F-Score & Accuracy for Nicad & CCCD

Tool	Source Example	Precision	Recall	F-Score	Accuracy
Nicad	**Apache**	.94	.89	.92	.99
	P-SQL	.12	1	.21	.96
	Python	.93	.5	.65	.97
	Avg.	**.66**	**.8**	**.59**	**.97**
MeCC	**Apache**	.94	.52	.67	.95
	P-SQL	.06	.4	.1	.95
	Python	.8	.5	.62	.97
	Avg.	**.6**	**.47**	**.46**	**.96**
Simcad	**Apache**	.67	.75	.71	.97
	P-SQL	0	-	0	.95
	Python	.8	.5	.62	.97
	Avg.	**.49**	**.63**	**.44**	**.96**
CCCD	**Apache**	1	.9	.95	.99
	P-SQL	.73	.96	.83	.98
	Python	.67	.84	.75	.98
	Avg.	**.83**	**.93**	**.88**	**.98**

All of the analyzed clone detection techniques were able to achieve a high rate of accuracy when examining open source applications. This is due to the relatively small number of clones that existed in these applications. While some of the tools fared better in specific areas, CCCD achieved overall better results overall. Nicad, MeCC, and Simcad all presented a lower precision and F-Score when searching for clones in P-SQL, possibly due the relatively large size of the analyzed subset of this application. When a statistically significant portion of this subset was created, many of the discovered clones were not selected for analysis, which helped lead to the smaller detection scores.

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 Discussion

During our analysis of concolic analysis for clone detection, we found several interesting areas that warrant further discussion. First, a discussion of the amount of time required to required to run the examined clone detection tools. Second, the effects that using various Levenshtein distance values have in determining clones has on the precision, recall, F-score, and accuracy values of concolic analysis for clone detection. Finally, a discussion on how the Levenshtein score between compared methods are relate to the likelihood of different types of clones found by concolic analysis.

4.1 Execution Times

One aspect of note in concolic analysis for clone detection is the amount of time required to search for clones on an application. Table 8 compares the execution time for Simian, Nicad, CloneDR, MeCC, Simcad, and concolic analysis in detection clones implemented using CCCD, on a small control file containing clones from Krawitz [30] and Roy *et al.* [42]. Run times were then compared against much larger applications including PostgreSQL 8.4.9, Python 2.5.1, and Apache 2.2.14. All comparisons were conducted on a Fedora 32-bit machine with a 2.5 GHz Intel Core 2 Duo Processor and 4 GB ram.

While the time required to find clones using concolic analysis has no effect on either its results or its ability to find clones, we do feel that this is a significant hurdle when employing this technique. Interestingly, the concolic analysis portion of the process is only a small percentage of the time required for clone discovery. When analyzing PostgreSQL, for example, the concolic analysis phase only took 1-2 seconds to complete. The round robin comparison portion consumes most of the analysis time largely due to the sheer number of comparisons which must take place. Future work may be done to reduce the number of comparisons in an effort to significantly speed up the clone detection process.

4.2 Levenshtein Distance

Concolic analysis for clone detection uses the Levenshtein distance algorithm to measure the similarity of two sets of concolic output, with sets of concolic output with a specific similarity scores marked as potential clones. Our first step in determining the most appropriate Levenshtein value was to use was to produce concolic output from the control, Apache, Python, and PostgreSQL applications, compare them against one another using a round robin methodology, then to record the Levenshtein distance scores. We then evaluated this against our clone oracle using Levenshtein scores of 0-40 with 5 point increments as a basis for determining clones. To obtain

Table 8: Execution Times

Source Example	Tool	Execution Time (seconds)
Control	**Simian**	.06
	Nicad	1
	CloneDR	3.6
	MeCC	1.9
	Simcad - Java	1.69
	Simcad - C	1.95
	CCCD	4.3
Apache	**Simian**	.46
	Nicad	1
	CloneDR	3.2
	MeCC	6.18
	Simcad	2.5
	CCCD	36
Python	**Simian**	.94
	Nicad	1
	CloneDR	6.6
	MeCC	8.11
	Simcad	3
	CCCD	98
PostgreSQL	**Simian**	.66
	Nicad	1
	CloneDR	1.8
	MeCC	7.2
	Simcad	3
	CCCD	51

the optimal number, we compared the precision, recall, F-score, and accuracy scores of each increment and found that for all of the codebases, the Levenshtein value of 30 produced the highest rates.

We combined the precision, recall, F-score and accuracy values of all four codebases and placed them into a chart 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 using our generated clone oracle using an aggregate of values from Apache, Python, and PostgreSQL.

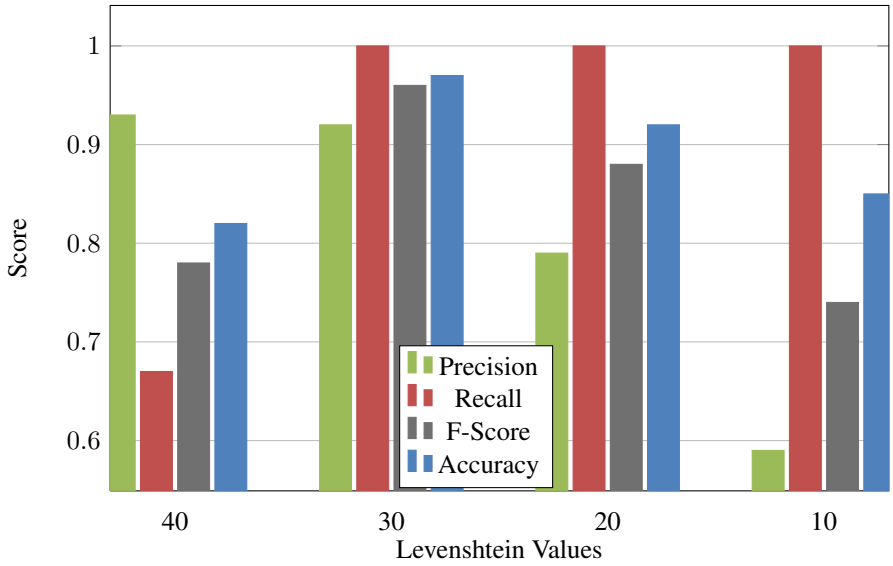

Fig. 2: Levenshtein Impact In Control Environment

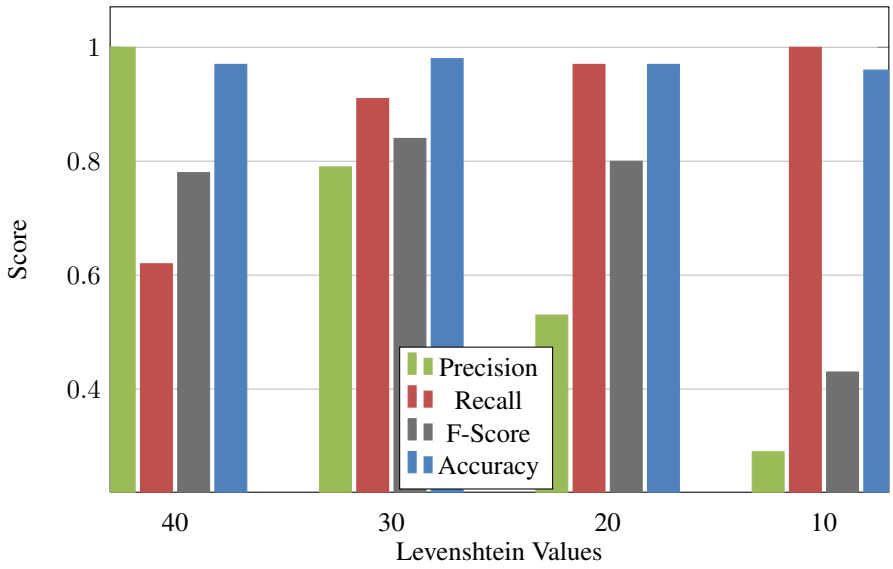

Fig. 3: Levenshtein Impact In Open Source Applications

In the controlled environment with the clones from Krawitz and Roy *et al.*, accuracy, recall, and precision values were collectively highest when a Levenshtein score of 30 was used. Similar results were observed when analyzing the open source ap-

plications for clones. A higher Levenshtein score is likely to discover more clones, but will also lead to more false positives, creating high precision but lower recall. Conversely, a low Levenshtein score will find fewer actual clones, but also have less false positives leading to high recall, but low precision. This is because a higher Levenshtein score means that the similarity threshold for noting cloned items will be reduced. A user could select different Levenshtein values depending on their desired levels of precision, recall, F-Score and accuracy.

These findings are important for several reasons. In both examples, the most appropriate Levenshtein score for attaining the highest accuracy, recall and precision was found to be 30, and this value has been used throughout our analysis. Additionally, these findings are indicative of those that future researchers may expect when using concolic analysis to find clones in their respective applications. In certain situations, researchers may wish to increase the recall or precision of their clone discovery technique, using resulting data to seek the most appropriate values.

4.3 Calculated Levenshtein Distance & Clone Types

An interesting discovery is how calculated Levenshtein distance may show indication of clone type. In general, a lower calculated distance score is indicative of a closer level of similarity between two compared items. Overall, the average similarity score for all types of clones was 21.77, while non-clones averaged 71.19. A smaller variation would likely lead to many more errors in the clone detection process. What is most interesting is that the average Levenshtein score between two compared methods may not only indicate if they are clones, but also may help to indicate what type of clone they are as well. These values were calculated by recording the Levenshtein distance for each type of identified clone in the oracle, along with items which were not clones. The scores were then averaged together; type-1 clones were found to have the lowest average, with more complicated clones showing progressively higher values. Final results are displayed in Figure 4.

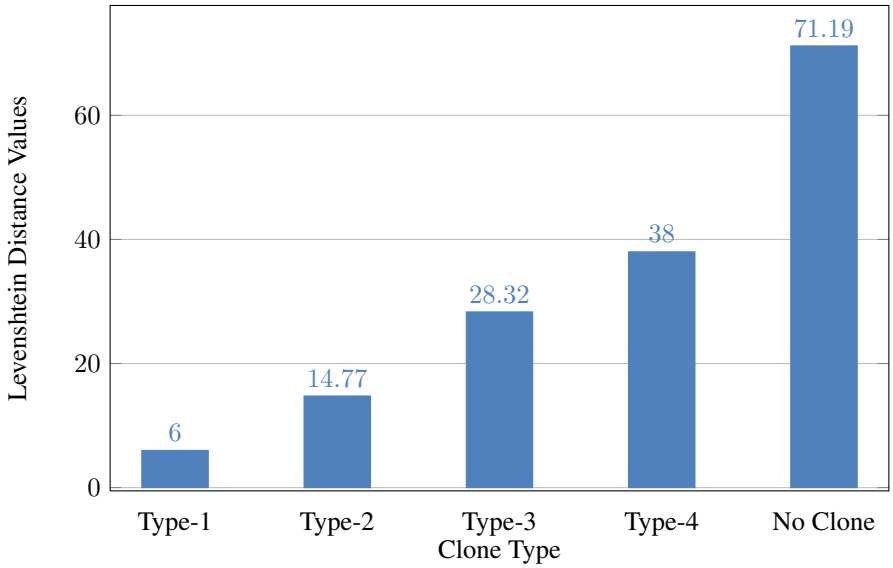

Fig. 4: Levenshtein Impact on Clone Types

5 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 [42]. Only two , however, 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 [27]. Krawitz [30] 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 also requires a substantial amount of random data, which may be a difficult and time consuming process to produce.

CCFinder is a powerful clone detection tool which has been extensively referenced in existing clone detection research [10, 18, 23]. The process used first transforms the input source text and then performs a token-by-token comparison in order to discover clones [22]. While this tool has been evaluated in a significant amount of previous research, all links to download the application from its website appear to be dead, so it was unable to be evaluated in our work.*[Dan says: Updated this sentence]*

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 [29, 45, 51]. Several tools exist for performing concolic analysis, including Crest [8], Java Path Finder [9], CUTE [45], and Pex [10].

Tempero [47] 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. Additionally, the clones are only from Java-based systems.

Lavoie and Merlo [33] created an clone oracle set containing type-3 clones using the Levensthein metric. There was no mention of type-4 clones being created as part of this oracle, and the provided oracle only contained Java code. Krawitz [30] and Roy *et al.* [42] both 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.

6 Threats to validity

There are certain threats to the validity of our results. First, 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, meaning that it will not be able to detect clones in non-functional portions of the software. Second, this technique is 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 application 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.

A significant portion of this study was based off previous research by Krawitz [30], Roy *et al.* [42] and Kim *et al.* [27]. Therefore, our results 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 is only one existing method known to reliably find type-4 clones. This makes it very difficult to test out a new mechanism in finding these clones specifically because there are very few benchmarks to be evaluated against. We are confident that concolic analysis is able to discover type-4 clones as is exemplified by our evaluation using the small sample oracle largely derived from Krawitz [30] and Roy *et al.* [42]. Unfortunately, since type-4 clones are very hard to manually identify and are only found by one existing

[8] http://code.google.com/p/crest/

[9] http://babelfish.arc.nasa.gov/trac/jpf/

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

tool, generating an accurate evaluation of a new technique in its ability to accurately identify type-4 clones is very difficult.

While we did our best to manually identify and classify clones using several people, and previous research has demonstrated the difficulty and problems with manually identifying and classifying code clones [50]. This indicates that other researchers may disagree with many of the clones identified and how they were classified in our work. This is a problem which is not at all unique to our work and is one that hinders other research as well [33].

There are also numerous clone detection tools that detect clones in numerous different 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. Many clone detection tools have adjustable inputs which may be altered to determine the size of the methods examined for clones, along with the similarity score needed to determine if two methods are defined as clones. While we did our best to use the most appropriate input settings for each tool, it is quite possible that more appropriate settings could have been selected to yield more accurate results. When we were unsure of the most appropriate setting, we chose to use the defaults for each tool.

7 Conclusion & Future Work

In the future, we plan on applying the techniques described in this paper to other areas of computing research. One area we will research is how type-4 clones affect software development including how problematic they actually are. While existing research has examined many of the effects that simpler clones have on the software development lifecyle [21], to our knowledge no work has been done to analyze the effect of type-4 clones specifically in that context.

Concolic analysis has only been evaluated in finding clones at the method level. However, many clones occur as only portions of methods, or across numerous methods. Future work is needed to determine the ability of the proposed technique in discovering clones at a more granular level or across methods.

Concolic Code Clone Detection represents a new and powerful clone detection technique. Concolic analysis executes various paths of an application. 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 is 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 [11]

[11] http://www.se.rit.edu/~dkrutz/CCCD/

References

1. Cccd concolic code clone detection. URL http://www.se.rit.edu/~dkrutz/CCCD/
2. Clonedr. URL http://www.semdesigns.com/Products/Clone/
3. Simian- similarity analyser:. URL http://www.harukizaemon.com/simian/
4. Mecc: Memory comparision-based clone detector (2013). URL http://ropas.snu.ac.kr/mecc/
5. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of the Second Working Conference on Reverse Engineering, WCRE '95, pp. 86–. IEEE Computer Society, Washington, DC, USA (1995). URL http://dl.acm.org/citation.cfm?id=832303.836911
6. Bard, G.V.: Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In: L. Brankovic, C. Steketee (eds.) Fifth Australasian Information Security Workshop (Privacy Enhancing Technologies) (AISW 2007), *CRPIT*, vol. 68, pp. 117–124. ACS, Ballarat, Australia (2007)
7. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance, ICSM '98, pp. 368–. IEEE Computer Society, Washington, DC, USA (1998). URL http://dl.acm.org/citation.cfm?id=850947.853341
8. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. Software Engineering, IEEE Transactions on **33**(9), 577–591 (2007). DOI 10.1109/TSE.2007.70725
9. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001). DOI 10.1109/2.962984. URL http://dx.doi.org/10.1109/2.962984
10. Choi, E., Yoshida, N., Ishio, T., Inoue, K., Sano, T.: Extracting code clones for refactoring using combinations of clone metrics. In: Proceedings of the 5th International Workshop on Software Clones, IWSC '11, pp. 7–13. ACM, New York, NY, USA (2011). DOI 10.1145/1985404.1985407. URL http://doi.acm.org/10.1145/1985404.1985407
11. Dang, Y., Zhang, D., Ge, S., Chu, C., Qiu, Y., Xie, T.: Xiao: tuning code clones at hands of engineers in practice. In: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12, pp. 369–378. ACM, New York, NY, USA (2012). DOI 10.1145/2420950.2421004. URL http://doi.acm.org/10.1145/2420950.2421004
12. Deissenboeck, F., Hummel, B., Juergens, E.: Code clone detection in practice. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp. 499–500. ACM, New York, NY, USA (2010). DOI 10.1145/1810295.1810449. URL http://doi.acm.org/10.1145/1810295.1810449
13. Duala-Ekoko, E., Robillard, M.P.: Clone region descriptors: Representing and tracking duplication in source code. ACM Trans. Softw. Eng. Methodol. **20**(1), 3:1–3:31 (2010). DOI 10.1145/1767751.1767754. URL http://doi.acm.org/10.1145/1767751.1767754

14. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99, pp. 109–. IEEE Computer Society, Washington, DC, USA (1999). URL http://dl.acm.org/citation.cfm?id=519621.853389
15. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Professional **2**(3), 17–23 (2000). DOI 10.1109/6294.846201. URL http://dx.doi.org/10.1109/6294.846201
16. Gold, N., Krinke, J., Harman, M., Binkley, D.: Issues in clone classification for dataflow languages. In: Proceedings of the 4th International Workshop on Software Clones, IWSC '10, pp. 83–84. ACM, New York, NY, USA (2010). DOI 10.1145/1808901.1808916. URL http://doi.acm.org/10.1145/1808901.1808916
17. Greenhill, S.J.: Levenshtein distances fail to identify language relationships accurately. Comput. Linguist. **37**(4), 689–698 (2011). DOI 10.1162/COLI_a_00073. URL http://dx.doi.org/10.1162/COLI_a_00073
18. Hotta, K., Sano, Y., Higo, Y., Kusumoto, S.: Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL '10, pp. 73–82. ACM, New York, NY, USA (2010). DOI 10.1145/1862372.1862390. URL http://doi.acm.org/10.1145/1862372.1862390
19. Jain, M., Benmokhtar, R., Jégou, H., Gros, P.: Hamming embedding similarity-based image classification. In: Proceedings of the 2nd ACM International Conference on Multimedia Retrieval, ICMR '12, pp. 19:1–19:8. ACM, New York, NY, USA (2012). DOI 10.1145/2324796.2324820. URL http://doi.acm.org/10.1145/2324796.2324820
20. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 672–681. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2486788.2486877
21. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 485–495. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ICSE.2009.5070547. URL http://dx.doi.org/10.1109/ICSE.2009.5070547
22. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7), 654–670 (2002). DOI 10.1109/TSE.2002.1019480. URL http://dx.doi.org/10.1109/TSE.2002.1019480
23. Kamiya, T., Ohata, F., Kondou, K., Kusumoto, S., Inoue, K.: Maintenance support tools for java programs: Ccfinder and jaat. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, pp. 837–838. IEEE Computer Society, Washington, DC, USA (2001). URL http://dl.acm.org/citation.cfm?id=381473.381749

24. Kapser, C.J., Godfrey, M.W.: Supporting the analysis of clones in software systems: Research articles. J. Softw. Maint. Evol. **18**(2), 61–82 (2006). DOI 10.1002/smr.v18:2. URL http://dx.doi.org/10.1002/smr.v18:2
25. Kawrykow, D.: Enabling Precise Interpretations of Software Change Data. McGill theses. McGill University Libraries (2011). URL http://books.google.com/books?id=WbhbnQEACAAJ
26. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: 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 (2013). DOI 10.1145/2377656.2377662. URL http://doi.acm.org/10.1145/2377656.2377662
27. Kim, H., Jung, Y., Kim, S., Yi, K.: Mecc: memory comparison-based clone detector. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 301–310. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1985835. URL http://doi.acm.org/10.1145/1985793.1985835
28. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. SIGSOFT Softw. Eng. Notes **30**(5), 187–196 (2005). DOI 10.1145/1095430.1081737. URL http://doi.acm.org/10.1145/1095430.1081737
29. Kim, Y., Kim, M., Kim, Y., Jang, Y.: 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, pp. 1143–1152. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337373
30. Krawitz, R.M.: Code clone discovery based on functional behavior. Ph.D. thesis, Nova Southeastern University (2012)
31. Krutz, D.E., Le, W.: A code clone oracle. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR '14. IEEE Press, Hyderabad, India (2014)
32. Krutz, D.E., Shihab, E.: Cccd: Concolic code clone detection. In: Reverse Engineering (WCRE), 2013 20th Working Conference on (2013). DOI 10.1109/WCRE.2012.60
33. Lavoie, T., Merlo, E.: Automated type-3 clone oracle using levenshtein metric. In: Proceedings of the 5th International Workshop on Software Clones, IWSC '11, pp. 34–40. ACM, New York, NY, USA (2011). DOI 10.1145/1985404.1985411. URL http://doi.acm.org/10.1145/1985404.1985411
34. Li, R.: 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, pp. 226–230. ACM, New York, NY, USA (2008). DOI 10.1145/1593105.1593164. URL http://doi.acm.org/10.1145/1593105.1593164
35. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. **32**(3), 176–192 (2006). DOI 10.1109/TSE.2006.28. URL http://dx.doi.org/10.1109/TSE.2006.28
36. Mondal, M., Roy, C.K., Schneider, K.A.: An empirical study on clone stability. SIGAPP Appl. Comput. Rev. **12**(3), 20–36 (2012). DOI 10.1145/2387358.2387360. URL http://doi.acm.org/10.1145/2387358.2387360

37. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165 – 1199 (2013). DOI http://dx.doi.org/10.1016/j.infsof.2013.01.008. URL http://www.sciencedirect.com/science/article/pii/S0950584913000323
38. Ros, M., Sutton, P.: 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, pp. 97–104. ACM, New York, NY, USA (2005). DOI 10.1145/1086297.1086311. URL http://doi.acm.org/10.1145/1086297.1086311
39. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY **115** (2007)
40. Roy, C.K., Cordy, J.R.: An empirical study of function clones in open source software. In: Proceedings of the 2008 15th Working Conference on Reverse Engineering, WCRE '08, pp. 81–90. IEEE Computer Society, Washington, DC, USA (2008). DOI 10.1109/WCRE.2008.54. URL http://dx.doi.org/10.1109/WCRE.2008.54
41. Roy, C.K., Cordy, J.R.: 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, pp. 172–181. IEEE Computer Society, Washington, DC, USA (2008). DOI 10.1109/ICPC.2008.41. URL http://dx.doi.org/10.1109/ICPC.2008.41
42. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009). DOI 10.1016/j.scico.2009.02.007. URL http://dx.doi.org/10.1016/j.scico.2009.02.007
43. Schulze, S., Apel, S., Kästner, C.: Code clones in feature-oriented software product lines. SIGPLAN Not. **46**(2), 103–112 (2010). DOI 10.1145/1942788.1868310. URL http://doi.acm.org/10.1145/1942788.1868310
44. Seaman, C.B.: Software maintenance: Concepts and practice. Journal of Software Maintenance and Evolution: Research and Practice **13**(2), 143–147 (2001). DOI 10.1002/smr.225. URL http://dx.doi.org/10.1002/smr.225
45. Sen, K., Marinov, D., Agha, G.: 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, pp. 263–272. ACM, New York, NY, USA (2005). DOI 10.1145/1081706.1081750. URL http://doi.acm.org/10.1145/1081706.1081750
46. Shukla, R., Misra, A.K.: Estimating software maintenance effort: a neural network approach. In: Proceedings of the 1st India software engineering conference, ISEC '08, pp. 107–112. ACM, New York, NY, USA (2008). DOI 10.1145/1342211.1342232. URL http://doi.acm.org/10.1145/1342211.1342232
47. Tempero, E.: Towards a curated collection of code clones. In: Proc. IWSC, pp. 53–59. IEEE (2013)
48. Uddin, M., Roy, C., Schneider, K.: Simcad: An extensible and faster clone detection tool for large scale software systems. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on, pp. 236–238 (2013). DOI 10.1109/ICPC.2013.6613857

49. Ueda, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Gemini: Maintenance support environment based on code clone analysis. In: Proceedings of the 8th International Symposium on Software Metrics, METRICS '02, pp. 67–. IEEE Computer Society, Washington, DC, USA (2002). URL http://dl.acm.org/citation.cfm?id=823457.824039
50. Walenstein, A., Jyoti, N., Li, J., Yang, Y., Lakhotia, A.: Problems creating task-relevant clone detection reference data. In: Proceedings of the 10th Working Conference on Reverse Engineering, WCRE '03, pp. 285–. IEEE Computer Society, Washington, DC, USA (2003). URL http://dl.acm.org/citation.cfm?id=950792.951349
51. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: Proceedings of the 2008 international symposium on Software testing and analysis, ISSTA '08, pp. 249–260. ACM, New York, NY, USA (2008). DOI 10.1145/1390630.1390661. URL http://doi.acm.org/10.1145/1390630.1390661
52. Yuan, Y., Guo, Y.: Cmcd: Count matrix based code clone detection. In: Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference, APSEC '11, pp. 250–257. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/APSEC.2011.13. URL http://dx.doi.org/10.1109/APSEC.2011.13
53. Zibran, M.F., Roy, C.K.: Ide-based real-time focused search for near-miss clones. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, pp. 1235–1242. ACM, New York, NY, USA (2012). DOI 10.1145/2231936.2231970. URL http://doi.acm.org/10.1145/2231936.2231970

A Appendix

We discussed precision, recall, F-score and accuracy in Section 3.4, where we omitted the results for several clone detection tools due to the inability to perform many of the calculations for specific tools since they were unable to find any clones. We present the complete results in Table 9, with incalculable values represented with an *x*.

Table 9: Precision, Recall, F-Score & Accuracy for Each Tool

Tool	Source Example	Precision	Recall	F-Score	Accuracy
Mecc-4	**Control**	0	x	x	.64
	Apache	1	.3	.46	.88
	P-SQL	0	0	x	.94
	Python	0	0	x	.97
CloneDR	**Control**	.78	1	.88	.67
	Apache	0	x	x	.95
	P-SQL	0	x	x	.95
	Python	0	x	x	.97
Simian	**Control**	.14	1	.26	.69
	Apache	.06	.33	.1	.94
	P-SQL	0	x	x	.94
	Python	0	x	?	.97
Nicad	**Control**	.59	1	.74	.85
	Apache	.77	1	.87	.96
	P-SQL	.03	1	.06	.95
	Python	.67	1	.8	.97
CCCD	**Control**	.92	1	.96	.97
	Apache	1	.9	.95	.99
	P-SQL	.73	.96	.83	.98
	Python	.67	.84	.75	.98