Intelligent Token-Based Code Clone Detection System for Large Scale Source Code

Abdulrahman Abu Elkhail
abdulrahman_abuelkh1@baylor.edu
Baylor University
Waco, Texas , 76706

Jan Svacina
jan_svacina2@baylor.edu
Baylor University
Waco, Texas , 76706

Tomas Cerny
tomas_cerny@baylor.edu
Baylor University
Waco, Texas , 76706

ABSTRACT

Fragments of source-code that are similar are known as code-clones and can cause many difficulties within software applications. As developers develop large-scale applications, code-clones can become more and more pervasive throughout the code-base. There are many proposed methods for detecting such clones in applications and in this paper, we present a novel method for code-clone detection in large-scale repositories. Our token-based code-clone detector, called Intelligent Clone Detection Tool (ICDT) can detect both exact and near-miss clones from large repositories. We present our method for detecting clones and then report the evaluation of ICDT using a large-scale code-clone benchmark, BigCloneEval. Lastly, we compare ICDT to other publicly available and state-of-the-art tools. We find that ICDT is more than capable of finding code-clones in large-scale repositories to a high degree of accuracy.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**;**Software maintenance tools**;**Formal software verification**;

KEYWORDS

Code Clone, Clone Detection, BigCloneBench, Case Study

ACM Reference Format:
Abdulrahman Abu Elkhail, Jan Svacina, and Tomas Cerny. 2019. Intelligent Token-Based Code Clone Detection System for Large Scale Source Code. In *Proceedings of International Conference on Research in Adaptive and Convergent Systems, Chongqing, China, September 24–27, 2019 (RACS '19),* 5 pages.
https://doi.org/10.1145/3338840.3355654

1 INTRODUCTION

Unchecked software development can lead to code-clones being introduced into software applications as developers grow the application through copying-and-pasting code. This can make maintaining such an application quite difficult [21, 36], especially in the case of software bugs. Without managing code-clones, the possibilities of bug pervasiveness through code-clones remains exceedingly high during the software development life-cycle. Another issue with code-clones is the overall size of the application which will grow larger than it should be as clones are introduced [5]. It can be easier for developers to use copied code instead of properly abstracting reusable components [36]. However, abstracting away reusable components mitigates the issue of duplicated source-code and also keeps the deliverable small [7] since when code is copied, the application grows by the size of the clone as well. Clearly it is necessary to keep the amount of code-clones within an application minimal and to document any clones that do arise within the application. However, keeping all clone information is a generally expensive process especially for a large and complex system. Therefore, in this paper, we propose a new token-based tool called ICDT in order to detect the code clones for a large scale software system. The ICDT tool extracts a list of a token sequence form the input source code through a lexical analyzer and applies the rule-based transformation to the sequence in order to detect clone code portions that have different syntax but have a similar meaning and to filter out code portions with specified structure patterns. Representing a source code as a token sequence enables us to detect clones with different line structures, which cannot be detected by the line-by-line algorithm.

The rest of the paper is organized as follows. Section 2 presents the basic definitions of code clones and clone types. Section 3 reviews the previous work related to this study. In Section 4, the proposed tool ICDT is presented in detail. It is followed by our description of various experiments conducted to evaluate the scalability of ICDT against state-of-the-art tools. Finally, the paper is concluded with threats to validity and future directions for research in this area.

2 BACKGROUND

Detection of code-clones is a well-documented area of research with tools and applications stretching back many decades. The high-level recognized clone classification is broken into two categories - syntactic and semantic clones. Syntactic clones refer to two code fragments which are similar based on their text [1, 24], while semantic clones are two code fragments similar based on their functions [9]. Furthermore, from the more detailed perspective, there are four types of code clones where the first three types fit the syntactic clone category and the fourth one fits the semantic clones.

Type-1: A type-1 code-clone is one in which the two fragments are exactly identical. However, the two code fragments do not need to be exactly the same with regards to whitespace, blanks, and comments as these are generally removed for the code-clone detection process.

RACS '19, September 24–27, 2019, Chongqing, China

ACM ISBN 978-1-4503-6843-8/19/09…$15.00
https://doi.org/10.1145/3338840.3355654

Type-2: A type-2 code-clone is one in which two code fragments are similar except for the renaming of some unique identifiers such as function/class names and variable identifiers. In a seminal paper on type-2 clones, Baker identifies the replacement of these unique identifiers as "parameterizing" the code fragment [2, 3].

Type-3: A type-3 code-clone is essentially a type-2 code-clone however the fragments may be modified. This includes adding and removing portions of the code from the two fragments or reordering statements within a code block.

Type-4: A type-4 code clone is different than the previous three in that a type-4 clone is *semantically* similar but not syntactically similar. These are much more difficult to find and generally code-clone detection tools either focus on types 1-3 or type 4.

Defining a code-clone is a difficult problem within itself, however, for the purposes of this paper we use a combination of well-accepted definitions of code clones and clone types [6, 22], as well as the definitions from a 2016 comprehensive study on code-clones [27]:

Code Fragment: A continuous segment of the source code, specified by (*l, s, e*), including the source file *l*, the line the fragment starts on, *s*, and the line it ends on, *e*.

Clone Pair: A pair of code fragments that are similar, specified by (*f1*, *f2*, ∅), including the similar code fragments *f1* and *f2*, and their clone type ∅.

Clone Class: A set of code fragments that are similar. Specified by the tuple (*f1, f2,. . . , fn,* ∅). Each pair of distinct fragments is a clone pair: (*fi, fj* , ∅), $i, j \in 1 \ldots n, i \neq j$.

Code Block: A sequence of code statements within braces.

3 LITERATURE REVIEW

Several techniques have been reported in the literature to detect the software clones. One of those techniques is a token-based technique that has been used in the CCFinder and the SourcererCC clone detection tools [12, 26]. The CCFinder tool which consists of the transformation of the input source text and a token-by-token comparison. Furthermore, it can extract code clones in C, C++, Java, COBOL, and other source files. The SourcererCC tool that can detect both exact and near-miss clones from large scale projects using a standard workstation. They proposed another token-based tool which is based on a filtering heuristic that reduces the number of token comparisons when the two code blocks are compared [25]. Another technique to detect the software clones is the abstract syntax tree which has been used to detect the exact tree matches; a number of adjustments are needed to detect equivalent statement sequences, commutative operands, and nearly exact matches [5, 15].

Several frameworks have been introduced in order to evaluate the clone detection tools [24]. Juergens et al and Svajlenko et al introduced the CloneDetective framework and Bellon's benchmark framework, respectively [11, 30]. Those frameworks are open source frameworks that have been used to evaluate the clone detection tools. Another framework to evaluate the clone detection tools is the BigCloneBench [31] which is a collection of eight million validated clones within IJaDataset-2.0, a big data software repository containing 25,000 open-source Java systems. BigCloneBench contains both intra-project and inter-project clones of the four primary clone types [31].

An approach has been proposed to examine if the differences present between the clones can be safely parameterized without causing any side-effects [32]. Another study has been presented in order to investigate whether a combination of clone detection and latent semantic indexing improves the detection of candidate re-implementations [4]. Another code clone search technique called Siamese has been used to improve clone search performance [20]. Kim et al proposed VUDDY which is capable of detecting security vulnerabilities in large software programs [14]. Roy and col. introduced NICAD [20] which clusters code clone candidates and then uses a set of techniques to extract actual clones [23], among them powerful source normalization which ensures high precision.

Learning-Based detection techniques have been used to detect the software clones. Matsushita et al present a novel algorithm for detecting clones by focusing on gaps by function applications [18]. Another learning-based detection technique has been used to detect the syntactic level clones based on deep learning algorithms [35].

A novel technique for detecting Android application clones have been presented [8, 16, 33]. Andarwin tool [8], WuKong tool [33], and SUIDroid tool [16] have been proposed in order to detect similar Android apps based on their semantic information. Another novel technique has been used to detect Android application clones based on the analysis of user interface (UI) information collected at runtime [28]. LibRadar tool [17] has been proposed to detect third-party libraries used in an Android app based on stable API features that are obfuscation resilient in most cases.

4 PROPOSED APPROACH

In this section, we present the clone detection process of a new token-based tool called ICDT. It can detect the code clones for a

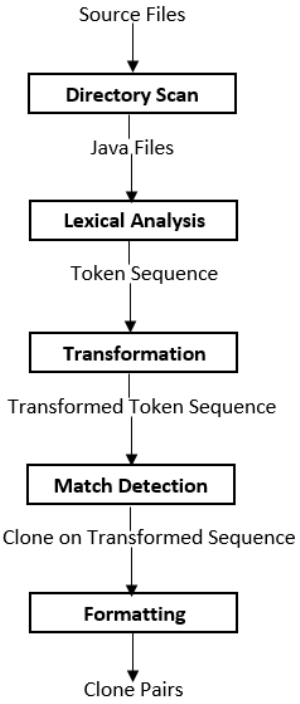

Figure 1: The clone detecting process of ICDT.

Table 1: Applied transformation rules.

Rule	Description
Remove package names	The package name is a word that begins with a small letter and `ClassName` is a capitalized word (`PackageName.ClassName`) ex. `java.lang.Math.sqrt()`. The transformation neglects the attribution so that they are considered equivalent in clone detection. For instance, `java.lang.Math.sqrt()` is transformed to `Math.sqrt()`.
Remove initialization lists	The initialization list is a sequence of Name, Number, Strings, and Operators. This rule is applied where the array is created with initialization by new expression. For example, `return new int[] {1,2,3};`
Remove accessibility keywords	This removes the accessibility keywords, e.g., `protected void functionName()` is transformed to `void functionName()`.
Separate class definition	Prevents extracting clone pairs of code portions that begin in the middle of one class and end in the middle of another.
Convert to a single block	Each statement is transformed to a single block. Ex. `if(x==true) foo=1;` is transformed to `if (x==true) {foo=1;}`

large scale software system. The clone detection process of ICDT is a process in which the input is source files and the output is clone pairs. Figure 1 shows the entire process of our token-based clone detecting technique. The process consists of five phases:

Phase-1 (Directory Scan): The directory scan is a utility class to list all java files in a directory.

Phase-2 (Lexical Analysis): In this phase, the Java Lexer analyzer has been used to divide each line in source files into tokens corresponding to a lexical rule of java. The tokens of all source files are concatenated into a single token sequence, so that finding clones in multiple files is performed in the same way as single file analysis. This removes the white spaces between tokens from the token sequence.

Phase-3 (Transformation): In this phase, the token sequence is transformed with two steps, the first step the token sequence is transformed, for instance, tokens are added, removed, or changed based on the transformation rules. Table 1 shows the transformation rules, Rule1, Rule2, and Rule3 have been used to remove the package name, initialization lists and accessibility keywords, respectively. Rule4 and Rule5 have been used to separate the class definition and to transform each statement into a single block. The second step is parameter replacement, in this step, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code portions with different variable names to become clone pairs. At the same time, the mapping information from the transformed token sequence into the original token sequences is stored for phase five.

Phase-4 (Match Detection): In this phase equivalent pairs from all the substrings on the transformed token sequence are detected as clone pairs. Each clone pair is represented as start and end flags which indicate to starting and ending of a clone pair.

Phase-5 (Formatting): In this phase, each location of the clone pair is converted into line numbers on the original source files.

Table 2: Number of clones found by ICDT

Clone Type	Description	# Clone Pairs
T1	Exact clones	35,787
T2	Renamed/Parameterized	4,573
VST3	Very strongly gapped clones	4,156
ST3	Strongly gapped clones	14,997
MT3	Moderately gapped clones	79,756
WT3/T4	Weakly gapped clones and semantic clones	7,729,291

5 THE PERFORMANCE EVALUATION

In this section, we evaluate the detection performance of ICDT in order to evaluate its scalability using a big benchmark of real clones. We measure its clone recall using the most recent benchmark, BigCloneBench [29, 31]. Clone recall is measured as a comparison between the output cluster (found code clones) and ground truth cluster (actual clones) [30].

BigCloneBench [29] is a clone detection benchmark consisting of manually validated clones in IJaDataset 2.0 [10], a big data source code repository containing 2.3 million Java source files (365MLOC) from 25,000 open-source projects. The benchmark was created, without the use of clone detection tools, by mining for functions implementing specific functionalities. Each clone pair is semantically similar to its target functionality and is one of the four primary clone types by their syntactical similarity. The published version of the benchmark considers 10 target functionalities [29]. There is no agreement on when a clone is no longer syntactically similar, therefore it is difficult to separate the Type-3 and Type-4 clones in BigCloneBench. Instead of that the Type-3 and Type-4 clones have been divided into four categories based on their syntactical similarity, as follows. Very Strongly Type-3 (VST3) clones have a syntactical similarity between 90% inclusive and 100% exclusive, Strongly Type-3 (ST3) in 70-90%, Moderately Type-3 (MT3) in 50-70% and Weakly Type-3/Type-4 (WT3/4) in 0-50%. Table 2 summarizes the number of clones in BigCloneBench per clone type.

We measure the recall of ICDT using BigCloneBench and compare it to four publicly available and state-of-the-art tools. A snapshot of the BigCloneBench benchmark with 43 target functionalities has been used for this study. We use only the clones that are at least 6 lines and 50 tokens in length. This is the standard minimum clone size for measuring recall [6, 31]. By specifying this both in lines and tokens we are able to configure the tools appropriately for clone size. Clone size is a primary clone detection configuration, and this prevents it from biasing the comparison of the tools' recall. We measure recall of ICDT with a 70% threshold. Table 3 shows the configurations of these tools for the experiment.

Table 3: Configuration of Code-Clone Detection Tools

Tool	Min length	Min similarity
ICDT	6 lines	70%
NiCad [23]	6 lines	70%
CloneDR [5]	6 lines	95%.
Jplag [19]	6 lines	1 character per line
CCfinder[13]	50 tokens	12 token types

Table 4: BigCloneBench Recall Results.

Tool	T1	T2	VST3	ST3	MT3	WT3/T4
ICDT	100	98	91	63	1	0
CCfinder	100	93	62	15	1	0
NiCad	100	100	100	95	1	0
CloneDR	100	94	71	21	1	0
Jplag	89	74	46	8	0	0

The recall measured by BigCloneBench is summarized in Table 4. It is summarized per clone type and per Type-3/4 category for all clones. ICDT has perfect detection of the Type-1 clones in BigCloneBench. It has near-perfect Type-2 detection. This shows that the 70% threshold is sufficient to detect the Type-2 clones without identifier normalizations. ICDT has excellent Type-3 recall for the VST3 category. ICDT Type-3 recall begins to drop off for the ST3 recall (63%). This is due to Type-3 clones having a higher incidence of Type-2 differences, causing them to not exceed ICDT 70% overlap threshold. Furthermore, ICDT does not normalize the identifier token names in order to maintain precision and index efficiency. Lowering the ICDT threshold would allow these to be detected, but could hurt precision. ICDT has a poor recall for the MT3 and WT3/T4 as these clones fall outside the range of syntactical clone detectors [31]. Compared to the competing tools, ICDT has the second-best recall overall, with NiCad taking the lead. Both tools have perfect Type-1 recall, and they have similar Type-2 recall, with NiCad taking a small lead. NiCad has better Type-3 recall due to its powerful source-normalization capabilities. CCfinder and ICDT tools have comparable Type-1 and Type-2 recall, with ICDT having the advantage of also detecting Type-3 clones. Jplag and CloneDR are the other competing clone detectors. Both ICDT and cloneDR have perfect Type-1 recall, but ICDT exceeds cloneDR in both Type-2 and Type-3 detection. Jplag has a poor overall recall for all clone types. ICDT has better recall and makes it an ideal choice for large-scale clone detection.

6 THREATS TO VALIDITY

Different configurations of the tools may result in a better or worse recall. Wang et al. [34] refer to this as the confounding configuration choice problem, and it is a challenge in all clone studies. However, we carefully experimented with its configurations to achieve appropriate results for our study. We used configurations that target the known properties of the benchmark, such as clone types and clone size. As for the other tools, we referred to the defaults and recommendations of the tools with respect to our knowledge of the benchmarks. This is the process a user would use to configure a tool for their own system, so our results reflect what a user should expect to receive. We did not execute the tools for various settings until an optimal result is found, as it is not possible for users to do this in practice. For the Type-3 clone detectors, lowering their thresholds would allow them to detect more clones in BigCloneBench [35]. However, the tools would have poor precision for low similarity thresholds.

7 CONCLUSION

In this paper, we introduced ICDT, a token-based clone detection tool, which can detect both exact and near-miss clones from large repositories using a standard workstation. We measure its recall using a state-of-the-art clone benchmark, the BigCloneBench. We find that ICDT is competitive with even the best of the state-of-the-art of Type-1, Type-2, and Type-3 clone detectors. Among tools using the token-based technique for detecting code clones, ICDT had exceptional results and compare to the clustering approach proved to be competitive. We believe that this technique can be improved on each stage (transformation, match detection, etc.) and bring even precise results in finding code clones. As future work, we are trying to extend the tool to accept source programs written in several programming languages at the same time.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Brenda S. Baker. 1993. A Theory of Parameterized Pattern Matching: Algorithms and Applications. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing (STOC '93).* ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/167088.167115

[3] Brenda S. Baker. 1996. Parameterized Pattern Matching. *J. Comput. Syst. Sci.* 52, 1 (Feb. 1996), 28–42. https://doi.org/10.1006/jcss.1996.0003

[4] V. Bauer, T. Völke, and S. Eder. 2016. Combining Clone Detection and Latent Semantic Indexing to Detect Re-implementations. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 3. 23–29. https://doi.org/10.1109/SANER.2016.26

[5] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM '98).* IEEE Computer Society, Washington, DC, USA, 368–. http://dl.acm.org/citation.cfm?id=850947.853341

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (Sep. 2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[7] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe. 2005. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Trans. Softw. Eng.* 31, 10 (Oct. 2005), 804–818. https://doi.org/10.1109/TSE.2005.114

[8] J. Crussell, C. Gibler, and H. Chen. 2015. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing* 14, 10 (Oct 2015), 2007–2019. https://doi.org/10.1109/TMC.2014.2381212

[9] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable Detection of Semantic Clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08).* ACM, New York, NY, USA, 321–330. https://doi.org/10.1145/1368088.1368132

[10] Ambient Software Evoluton Group. 2013. IJaDataSet 2.0. (2013). https://sites.google.com/site/asegsecold/projects/seclone

[11] E. Juergens, F. Deissenboeck, and B. Hummel. 2009. CloneDetective - A workbench for clone detection research. In *2009 IEEE 31st International Conference on*

Software Engineering. 603–606. https://doi.org/10.1109/ICSE.2009.5070566

[12] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[13] T. Kamiya, F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue. 2001. Maintenance support tools for Java programs: CCFinder and JAAT. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* 837–838. https://doi.org/10.1109/ICSE.2001.919197

[14] S. Kim, S. Woo, H. Lee, and H. Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[15] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. IEEE Computer Society, Washington, DC, USA, 253–262. https://doi.org/10.1109/WCRE.2006.18

[16] F. Lyu, Y. Lin, J. Yang, and J. Zhou. 2016. SUIDroid: An Efficient Hardening-Resilient Approach to Android App Clone Detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 511–518. https://doi.org/10.1109/TrustCom.2016.0104

[17] Z. Ma, H. Wang, Y. Guo, and X. Chen. 2016. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 653–656.

[18] Tsubasa Matsushita and Isao Sasano. 2017. Detecting Code Clones with Gaps by Function Applications. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2017)*. ACM, New York, NY, USA, 12–22. https://doi.org/10.1145/3018882.3018892

[19] Lutz Prechelt and Guido Malpohl. 2003. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science* 8 (03 2003).

[20] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* 24 (03 2019). https://doi.org/10.1007/s10664-019-09697-7

[21] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165 – 1199. https://doi.org/10.1016/j.infsof.2013.01.008

[22] Chanchal Kumar Roy and James R. Cordy. 2007. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541, Queen's University* 115 (2007).

[23] C. K. Roy and J. R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[24] C. K. Roy and J. R. Cordy. 2009. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. 157–166. https://doi.org/10.1109/ICSTW.2009.18

[25] H. Sajnani and C. Lopes. 2013. A parallel and efficient approach to large scale clone detection. In *2013 7th International Workshop on Software Clones (IWSC)*. 46–52. https://doi.org/10.1109/IWSC.2013.6613042

[26] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[27] Abdullah Sheneamer and Jugal Kalita. 2016. A Survey of Software Clone Detection Techniques. *International Journal of Computer Applications* 137 (03 2016), 1–21. https://doi.org/10.5120/ijca2016908896

[28] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang. 2015. Detecting Clones in Android Applications through Analyzing User Interfaces. In *2015 IEEE 23rd International Conference on Program Comprehension*. 163–173. https://doi.org/10.1109/ICPC.2015.25

[29] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

[30] J. Svajlenko and C. K. Roy. 2014. Evaluating Modern Clone Detection Tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 321–330. https://doi.org/10.1109/ICSME.2014.54

[31] J. Svajlenko and C. K. Roy. 2015. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 131–140. https://doi.org/10.1109/ICSM.2015.7332459

[32] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. 2015. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering* 41, 11 (Nov 2015), 1055–1090. https://doi.org/10.1109/TSE.2015.2448531

[33] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 71–82. https://doi.org/10.1145/2771783.2771795

[34] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 455–465. https://doi.org/10.1145/2491411.2491420

[35] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[36] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2015. Classification model for code clones based on machine learning. *Empirical Software Engineering* 20 (08 2015). https://doi.org/10.1007/s10664-014-9316-x