Towards a Big Data Curated Benchmark of Inter-Project Code Clones

Jeffrey Svajlenko*, Judith F. Islam*, Iman Keivanloo†, Chanchal K. Roy*, Mohammad Mamun Mia*
*Department of Computer Science, University of Saskatchewan, Canada
{jeff.svajlenko, judith.islam, chanchal.roy}@usask.ca, md.mamunmia@gmail.com
†Department of Electrical and Computer Engineering, Queen's University, Canada
iman.keivanloo@queensu.ca

***Abstract*—Recently, new applications of code clone detection and search have emerged that rely upon clones detected across thousands of software systems. Big data clone detection and search algorithms have been proposed as an embedded part of these new applications. However, there exists no previous benchmark data for evaluating the recall and precision of these emerging techniques. In this paper, we present a big data clone detection benchmark that consists of known true and false positive clones in a big data inter-project Java repository. The benchmark was built by mining and then manually checking clones of ten common functionalities. The benchmark contains six million true positive clones of different clone types: Type-1, Type-2, Type-3 and Type-4, including various strengths of Type-3 similarity (strong, moderate, weak). These clones were found by three judges over 216 hours of manual validation efforts. We show how the benchmark can be used to measure the recall and precision of clone detection techniques.**

I. Introduction

Historically, clone detection has focused on finding duplicate code within a software system in order to cancel out the effects of ad-hoc code reuse (e.g., copy and paste) [1]. Recently, new applications for clone detection and search have emerged relying on detected clones among a large number of software systems. Since classical clone detection tools do not support the needs of such emerging applications, new large-scale clone detection algorithms are being proposed as an embedded part of the emerging applications. For example, big data clone detection and clone search is used to find similar mobile applications [2], intelligently tag code snippets [3], find code examples [4], and so on. A big data code clone benchmark is needed to evaluate and compare the effectiveness of the underlying clone detection or clone search process of such emerging applications and also general-purpose big data clone detection algorithms (e.g., [5], [6], [7]).

Benchmarks exist for classical clone detection tools, which scale to a single system or a small repository. The most notable benchmark was created by Bellon et al. [8] by validating 2% of the union of six clone detectors for eight subject systems, a task that required 77 hours of manual efforts. Krutz and Le [9] followed a similar approach with modern tools and used a more rigorous validation process, including the use of several judges. While their data has high confidence, their benchmark is small, only 66 method clone pairs. In summary, the common approach to existing benchmarks is to mine clones using clone detectors. The problem with this approach is the benchmark only includes the clones the participating tools are able to detect. This gives the tools used to build the benchmark an unfair advantage over tools that did not contribute to the benchmark. The clones the participating tools failed to detect create a gap in the benchmark, resulting in the over-estimation of recall. Studies by Baker [10], and Svajlenko and Roy [11] have found several problems in Bellon's benchmark and the union method in general. Furthermore, existing benchmarks contain few clones of any particular code snippet, which makes it difficult to evaluate clone search algorithms. Alternatively, the Mutation Framework [12] could be adapted to big data, but a benchmark of real clones is still needed for thorough evaluation of big data clone detection algorithms.

In this paper, we introduce a benchmark of known true and false clones mined from the big data inter-project repository IJaDataset 2.0 [13] (25,000 subject systems, 365MLOC). We call this benchmark BigCloneBench and we built it, without the use of clone detectors, by mining IJaDataset for clones of frequently implemented functionalities. We used search heuristics to automatically identify code snippets in IJaDataset that might implement a target functionality. These candidate snippets are manually tagged as true or false positives of the target functionality by judges. The benchmark is populated with the true and false clones discovered by the tagging process. We typify each of these clones and measure their syntactical similarity. The current version of BigCloneBench covers ten functionalities (a.k.a. cases) including 6 million true clone pairs and 260 thousand false clone pairs. The benchmark can be found at the following URL: github.com/clonebench/BigCloneBench.

Our contribution is a big data benchmark that can be used to measure the recall and precision of big data clone detection and clone search techniques. This evaluation is needed to support the development and improvement of emerging big data clone detection and search algorithms. Since we mined clones of particular functionalities, our benchmark is also ideal for evaluating semantic clone detectors. To the best of our knowledge, there exists no benchmark for semantic clone detection. The benchmark can be used to evaluate classical clone detectors by evaluating them for subsets of the benchmark within their scalability constraints. Our benchmark consists of a larger variety of clones than can be found in a benchmark spanning only a handful of subject systems. Our data may also be useful for studying semantic and syntactic duplication across the open-source Java development community.

II. Background

(Code) Snippet: A continuous segment of source code. Specified by the triple (l, s, e), including the source file l, the

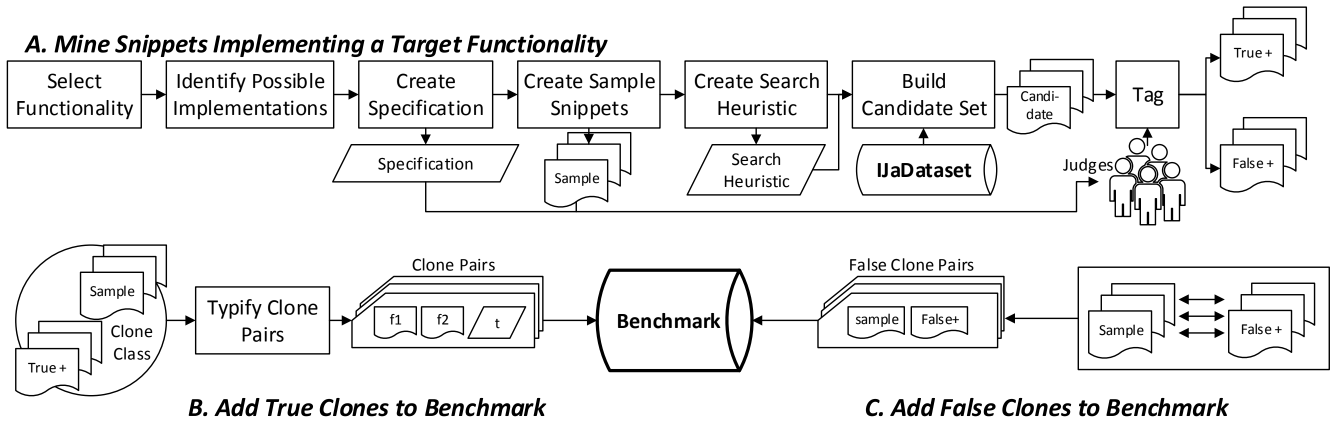

Fig. 1. Methodology

line the snippet starts on, s, and the line it ends on, e.

Clone (Pair): A pair of code snippets that are similar. Specified by tuple (f_1, f_2, ϕ), the similar code snippets f_1 and f_2, and the criterion of their similarity ϕ.

Clone Class: A set of code snippets that are similar. Specified by tuple $(f_1, f_2, ..., f_n, \phi)$. Each pair of distinct snippets is a clone pair: $(f_i, f_j, \phi),\ i, j \in 1..n,\ i \neq j$.

Researchers agree upon four primary clone types [1], [8]. The clone types are mutually exclusive, and are defined by the clone detection capabilities needed to identify them.

Type-1: Syntactically identical code snippets, except for differences in white space, layout and comments.

Type-2: Syntactically identical code snippets, except for differences in identifier names, literal values, white space, layout and comments.

Type-3: Syntactically similar code snippets that differ at the statement level. Snippets have statements added, modified and/or removed with respect to each other.

Type-4: Syntactically dissimilar code snippets that implement the same functionality.

Clone detection techniques are commonly evaluated using the information retrieval metrics *recall* and *precision*. Recall is the ratio of the clones within a source repository that a detector is able to detect. Precision is the ratio of the candidate clones reported by a detector which are true clones not false clones.

III. Methodology

Our benchmark consists of clones of common functionalities in IJaDataset. We built this benchmark by mining for code snippets that implement candidate functionalities. This process enables us to identify true and false clones of the four primary clone types. We begin by selecting a functionality that is commonly needed in open source Java software systems, and identify how it may be implemented in Java. Using these implementations, we design a search heuristic to locate snippets in IJaDataset that might implement the functionality. These candidate snippets are manually tagged by judges as true or false positives of the target functionality. Using the tagging data, we identify true and false clone pairs, which we add to the benchmark after identifying their clone types. This process is repeated for additional target functionalities, creating a live benchmark that improves over time. A single iteration of this process is shown in Figure 1. Iterations of this process can be performed in parallel. This methodology can be executed for any snippet granularity (method, block, etc). We target the method granularity as it is the granularity supported by the most tools, and because methods nicely encapsulate functionalities.

A. Snippet Tagging

In this stage we tag code snippets of IJaDataset as true or false positives of a target functionality (e.g., bubble sort). IJaDataset contains 24 million (method granularity) snippets, which is too many to inspect manually. So we construct a search heuristic to identify the snippets that might implement a given functionality. The tagging process occurs over seven distinct steps as follows. In the following, we use the bubble sort case (one of our ten cases) as a running example.

(1) Select Target Functionality. We begin by selecting a commonly needed functionality in open-source Java projects as our *target functionality*. It is likely that IJaDataset will contain a number of snippets that implement this functionality. For example, we chose the functionality bubble sort. While not the best sorting algorithm, it is well known and easy to understand and implement, and is therefore likely to appear in IJaDataset.

(2) Identify Possible Implementations. To create a search heuristic, we identify how the target functionality can be implemented in Java. We review Internet discussion (e.g., Stack Overflow) and API documentation (e.g., JavaDoc) to identify the common implementations of the target functionality. These resources are frequently used by open source developers, so we expect similar implementations to appear in IJaDataset. Implementations may use different support methods and classes (APIs), or they may express the algorithms using different syntax (e.g., different control flow statements). We perform this process incrementally, beginning with the most popular implementations. For example, bubble sort is typically implemented for array data structures in Java. The implementations vary in data comparison logic, which depends on the type of data being stored, and alternate ways of expressing the algorithm in syntax, particularly control flow statements.

(3) Create specification. We create a *specification* of the functionality, including the minimum steps or features a snippet must realize to be a true positive of the target functionality.

```
public static void bubbleSort(int[] arr) {
  int last_exchange;
  int right_border = arr.length - 1;
  do {
    last_exchange = 0;
    for (int j = 0; j < arr.length - 1; j++) {
      if (arr[j] > arr[j + 1]){
        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
        last_exchange = j;
      }
    }
    right_border = last_exchange;
  } while (right_border > 0);
}
```

Fig. 2. Sample Snippet: Bubble Sort

For the bubble sort case, we decided our specification would be "implement bubble sort for a linear data structure". Any snippet meeting this specification is a true positive.

(4) Create Sample Snippet. For each identified implementation, we collect a sample snippet that uses the implementation to achieve the functionality. The sample snippet is sample code found during our research of the implementation (Step 2). The sample implementation is a minimum working code example, and includes only the steps and features that are part of our specification of the target functionality (Step 3). The sample snippets are added to IJaDataset as additional crawled open-source code. For example, we added to IJaDataset six sample snippets that implement bubble sort (e.g., Figure 2) using variations in data comparison and control flow logic. These are open-source implementations found in Internet discussions.

(5) Create Search Heuristic. We create a search heuristic to locate snippets in IJaDataset that might implement the target functionality. Drawing from the identified possible implementations, the specification, and the sample snippets, we identify the keywords and source code patterns that are intrinsic to the identified implementations of the functionality. We expect snippets that implement the functionality to contain some combination of these keywords and source patterns. These keywords and source patterns are implemented as regular expression matches and are combined into a logical expression. Keywords and patterns that are expected to appear together in an implementation are 'AND'ed, while groups of keywords and patterns that are from different implementations are 'OR'ed. We design the heuristic to locate a large sample of the functions implementing the target functionality without identifying too many false positives that the judges are overburdened.

For example, the implementations of bubble sort do not rely upon any particular class or method from the Java standard library which could be used as a keyword. Also, implementations may not be conveniently named "BubbleSort", while the keyword "Sort" is too generic and returns many false positives. Data comparison and control flow logic may vary between implementations, so they are not good source patterns to base the heuristic on. Common to all implementations is the need to swap adjacent elements in the data array. We experimented with various source patterns related to this step of the algorithm and found that source patterns such as 'arr[j] = arr[j+1]' found many bubble sort implementations without over-burdening the judge with too many false positives. We used it as our search heuristic with the regular expression allowing any alternate identifier names.

(6) Build Candidate Set. The search heuristic is executed for every snippet in IJaDataset to identify possible *candidate snippets* that might implement the target functionality. For example, the search heuristic revealed 551 candidate snippets for bubble sort.

(7) Manual Tagging. Judges manually tag all of the candidate snippets as true or false positives of the target functionality. The judges are provided the specification and the sample snippets. They are instructed to tag any snippets that meets the specification as a true positive. If multiple judges are used, then disagreement is settled by the judgement of the majority. Therefore an odd number of judges is required, even if the final judge only tags the snippets under contention. For example, manual tagging found 163 of the candidates implemented bubble sort. The remaining 388 were false positives.

The final result of the snippet tagging stage is a set of true positives (snippets that implement the target functionality) and a set of false positives (snippets that do not implement the target functionality).

B. True Clone Pairs

From the tagging process, we know that the true positive snippets (validated in Step 7) and sample snippets (specified in Step 4) of a target functionality form an oracled true clone class of snippets that implement the same functionality. If we have p true positive snippets and s sample snippets for a target functionality, a true clone class of size $s+p$, then this results in $(s+p)(s+p-1)/2$ oracled true clone pairs of that functionality in IJaDataset. Each of these clone pairs implement the target functionality, and are one of the four primary clone types. For the bubble sort case, tagging found 163 snippets that implement bubble sort, and we added 6 sample snippets that implement bubble sort. This is a clone class of 169 snippets implementing bubble sort, or a total of 15,576 clone pairs. All of these snippets are semantically similar, while some are also syntactically similar.

To enrich the benchmark with meta-data, we typify the clones and measure their syntactical similarity. A clone pair can be typified as Type-1 or Type-2 if the snippets become textually identical after the appropriate source normalization. Type-1 normalization includes removing comments and a strict pretty-printing. Type-2 normalization expands this to include the systematic renaming of identifiers, and the replacing of literals with default values (e.g., numerics to 0, strings to "default", etc). If the snippets are not identical after these normalizations then, because they implement the same functionality, they are either Type-3 or Type-4. For such cases we examine and report their syntactical similarity.

We measure the syntactical similarity of the clones using a line-based metric after full normalization. This includes the removal of comments, a strict pretty printing, the renaming of all identifiers to a common value (e.g., 'X') and the change of all literal values to a common value (e.g., '0'). This blind normalization of identifiers and literals is needed as their values will not align due to changes at the statement level. The similarity metric measures the minimum ratio of the lines one snippet shares with another after normalization.

These clone pairs implement the same functionality, so they are Type-3 if they are also syntactically similar, or Type-4 if they are syntactically dissimilar. However, there is no consensus on the minimum similarity of a Type-3 clone, so it is difficult to separate the Type-3 and Type-4 clones in our case. We accept this ambiguity and instead divide the Type-3 and Type-4 into three categories based on their syntactical similarity values: Strongly Type-3, similarity in range $[0.7, 1.0)$, Moderately Type-3, $[0.5, 0.7)$, and Weakly Type-3+4, $[0.0, 0.5)$. This division by similarity is useful for measuring a tool's recall for different contexts.

We define strongly Type-3 clones as those that are at least 70% similar at the statement level. This is the region we expect most syntactical detectors to operate in. These clones are very similar, but contain some statement-level differences. The moderately Type-3 clones share at least half of their syntax, but contain a significant amount of statement-level differences. Syntactical clone detectors typically do not operate in this range because there is a higher chance that code snippets with only 50-70% shared syntax are only coincidentally similar. Detectors may need some level of semantic awareness to operate in this similarity region without diminished precision. We define the clones that share less than 50% of their syntax as weakly Type-3 or Type-4 clones.

C. False Clone Pairs

From manual inspection (Step 7) we know that the snippets tagged as false positives of a target functionality do not implement the target functionality, while the sample snippets (Step 4) implement only the target functionality. So each pair of sample snippet and false positive tagged snippet for a target functionality is an oracled false clone pair in our benchmark. If we have s sample snippets for a target functionality, and found f false positives snippets of this functionality, then this results in sf false clone pairs. Tagging revealed 398 false positive snippets that do not implement bubble sort. We added six sample snippets that only implement bubble sort. This is 5432 false clone pairs. While the false clones might share some syntactical similarity, our manual tagging process has validated that this similarity is coincidental.

IV. Data Summary

For the creation of our benchmark, 60 thousand snippets were tagged across 10 distinct functionalities, an effort that required 216 hours of manual tagging by three judges. From the tagging data, we were able to identify 6 million true clone pairs (Section III-B) and 260 thousand false clone pairs (Section III-C) in IJaDataset. Our tagging data and benchmark contents are summarized per functionality in Table I.

The left side of the table summarizes our tagging data and efforts per functionality. Including, the number of sample snippets added to IJaDataset per functionality, s, the number of IJaDataset snippets tagged as as true , p, or false positives, f, of the functionality, and the number of hours invested in tagging the snippets. Tagging hours includes only those spent by the judges actively tagging the final data. It does not include time spent researching the functionality and designing its search heuristic, rest periods taken by the judges, or time the judges spent tagging training data to prepare for the final data.

The right side of the table summarizes the number of true and false clone pairs discovered by the snippet tagging efforts. Each of the true clone pairs listed is a clone of the listed functionality. The tagging efforts locate $\frac{1}{2}(s+p)(s+p-1)$ oracled clone pairs per functionality, as described in Section III-B. The Type-3 and Type-4 clones are divided by their syntactical similarity as described in Section III-B. Each of the false clone pairs listed is a pair of code snippets that do not share functionality. The tagging efforts locate sf oracled false clone pairs as described in Section III-C.

V. Evaluating Clone Detectors

Our benchmark can be used to measure the recall and precision of big data clone detection techniques (e.g., [5]). Recall can be measured as in (1), where B_{tc} is the set of all true clone pairs in our benchmark, and D is the set of clone pairs reported by the detector. Perhaps more interesting is measuring the detectors' recall of subsets of B_{tc}. For example, all clones of a particular functionality, all clone of a particular type, all Type-3 clones within a particular range of syntactical similarity, etc. Or even all clones of a specific sample function for evaluating clone search tools.

$$recall = \frac{D \cap B_{tc}}{B_{tc}} \qquad (1)$$

The benchmark can be used to measure an upper and lower bound on a detector's precision, as shown in (2), where B_{fc} is the false clone pairs in the benchmark. The lower bound calculation assumes every detected clone pair unknown to the benchmark is a false clone, while the upper bound assumes the unknown clone pairs are true clones. This measured range in precision can be quite wide. We therefore provide a estimate of precision as in (3). This estimates precision as the ratio of the known clones pairs (true and false) found by the detector that are true clones. It ignores detected clones that are unknown to the benchmark.

$$\lfloor precision \rfloor = \frac{|D \cap B_{tc}|}{D} \quad \lceil precision \rceil = 1 - \frac{|D \cap B_{fc}|}{|D|} \qquad (2)$$

$$precision = \frac{|D \cap B_{tc}|}{|D \cap (B_{tc} \cup B_{fc})|} \qquad (3)$$

VI. Applications of BigCloneBench

While the primary use case of our benchmark is to evaluate big data clone detectors, it may also be used to evaluate other classifications of clone detectors. The benchmark is ideal for evaluating clone search algorithms as the benchmark contains many clones of a target snippet (the sample snippets). While classical detectors cannot be executed for IJaDataset in its entirety, they could be evaluated for subsets of the benchmark. The subsets would need to be small enough such that the tool could be executed for the relevant files without scalability issues. The subsets could be randomly chosen, could be all the true and false clones found for a functionality, or could even be the clones found in one of the 25,000 original subject systems crawled for IJaDataset. High confidence could be achieved by evaluating the classical tool for a large number of subsets. The primary advantage of using our big data benchmark to

TABLE I. SNIPPET TAGGING AND BENCHMARK DATA SUMMARY

Functionality	**Tagging Data**				**Oracled Clone Pairs from Tagging**					
	Sample	**Tagged Snippets**		**Tagging**	**True Clone Pairs, $(s+p)(s+p-1)/2$**					**False Clone**
	Snippets, s	**True+, p**	**False+, f**	**Hours**	**T1**	**T2**	**Strongly T3**	**Moderately T3**	**Weakly T3 & T4**	**Pairs, sf**
Web Download	3	910	12,946	50	1,554	9	1,439	2,715	410,611	38,838
Secure Hash (MD5)	1	1,342	4,564	21	632	587	3,294	24,923	871,717	4,564
Copy a File	6	3,084	34,018	134	13,805	3,116	5,947	24,199	4,725,438	204,108
Decompress Zip	2	7	28	0.2	0	0	1	1	34	56
FTP Authenticated Login	11	304	382	2.5	9	0	94	191	49,161	4,202
Bubble Sort	14	163	388	2	43	4	239	1,752	13,538	5,432
Init. SGV With Model	1	23	78	0.4	3	7	5	2	259	78
SGV Selection Event Handler	1	10	1,272	4.6	0	0	0	0	55	1,272
Create Java Project (Eclipse)	1	22	0	0.25	0	0	8	0	245	0
SQL Update and Rollback	2	135	12	0.6	122	10	259	97	8,828	24
Total	**42**	**6,000**	**53,688**	**216**	**16,168**	**3,733**	**11,286**	**53,880**	**6,079,886**	**258,574**
		59,688			**6,164,953**					

T3 Categories by Syntax Similarity Ranges - Strongly: [0.70,1.0) **Moderately**: [0.50,0.70), **Weakly**: [0.0,0.50) SGV = ScrollingGraphicalViewer (Eclipse GEF API)

evaluate classical tools is clone variety. Classical benchmarks only consider 1-10 subject systems, which provide a limited variety of clones, especially Type-1 and Type-2. In contrast, our benchmark considers 25,000 subject systems.

Since our benchmark consists of clones of particular functionalities, it is very useful for evaluating semantic clone detectors. To our knowledge, there is also no benchmark for semantic clone detectors. While semantic clone detectors may not be scalable to big data, they could be executed for subsets of the benchmark. Good subsets would be the individual functionalities, or a random selection of true and false positive clones from each of the functionalities.

VII. THREATS TO VALIDITY

Our methodology requires multiple judges to reduce subjectivity and human error during snippet tagging. At present we have used only one judge per tagged functionality. We minimized subjectivity by creating a clear specification of the requirements of a true positive snippet for each target functionality. We are currently revisiting these functionalities with two additional judges to reduce bias.

When building the search heuristics we targeted only the most popular implementations of a functionality. We have likely missed some snippets in IJaDataset that implement the functionality. We are currently prioritizing the expansion of the benchmark to additional functionalities to improve its variety. As future work we can return to the previously tagged functionalities and expand their search hueristic to include less popular approaches, and tag the missed snippets.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented a curated benchmark of inter-project clones in IJaDataset, a big data source code repository. This benchmark was created using a novel functionality-based clone mining approach. Unique to our benchmark is the identification of both semantically and syntactically similar clones. Unlike previous clone mining efforts (e.g., [8], [9]), our benchmark was built independently of clone detection tools, which is the recommended approach for evaluating modern clone detection tools [11]. This means it is not biased or limited to the clones that detectors are able to locate. This makes the benchmark ideal for identifying weaknesses in the current detection techniques.

For the next release of BigCloneBench, we plan to expand the benchmark with additional functionalities, and to increase the confidence of the data with additional judges. To achieve a live benchmark, we plan to open our process and tools to the community so that tagging can be achieved in a collaborative way. As future work, we plan to work towards better separation of the Type-3 and Type-4 clones. We also plan to expand the benchmark to additional languages and clone granularities.

REFERENCES

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, 2009.

[2] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE*, 2014, pp. 175–186.

[3] J.-w. Park, M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Surfacing code in the dark: an instant clone search approach," *Knowl. and Inf. Syst.*, pp. 1–33, 2013.

[4] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *ICSE*, 2014.

[5] J. Svajlenko, I. Keivanloo, and C. K. Roy, "Big data clone detection using the classical detectors: An exploratory study," *Journal of Soft. Evol. and Process*, 2014, 35 pp., DOI: 10.1002/smr.1662 (in press).

[6] H. Sajnani and C. Lopes, "A parallel and efficient approach to large scale clone detection," in *IWSC*, 2013, pp. 46–52.

[7] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *CSMR*, 2012, pp. 309–318.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 577–591, 2007.

[9] D. E. Krutz and W. Le, "A code clone oracle," in *MSR*, 2014, pp. 388–391.

[10] B. Baker, "Finding clones with dup: Analysis of an experiment," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 608–621, 2007.

[11] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *ICSME*, 2014, 10 pp.

[12] J. Svajlenko, C. Roy, and J. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *IWSC*, 2013, pp. 8–9.

[13] Ambient Software Evoluton Group, "IJaDataset 2.0," http://secold.org/projects/seclone, January 2013.