Evaluating Clone Detection Tools with BigCloneBench

Jeffrey Svajlenko Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
jeff.svajlenko@usask.ca chanchal.roy@usask.ca

***Abstract*—Many clone detection tools have been proposed in the literature. However, our knowledge of their performance in real software systems is limited, particularly their recall. In this paper, we use our big data clone benchmark, BigCloneBench, to evaluate the recall of ten clone detection tools. BigCloneBench 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. We use this benchmark to evaluate the recall of the tools per clone type and across the entire range of clone syntactical similarity. We evaluate the tools for both single-system and cross-project detection scenarios. Using multiple clone-matching metrics, we evaluate the quality of the tools' reporting of the benchmark clones with respect to refactoring and automatic clone analysis use-cases. We compare these real-world results against our Mutation and Injection Framework, a synthetic benchmark, to reveal deeper understanding of the tools. We found that the tools have strong recall for Type-1 and Type-2 clones, as well as Type-3 clones with high syntactical similarity. The tools have weaker detection of clones with lower syntactical similarity.**

I. Introduction

Clone detection tools locate similar source code within or between software systems. Instances of similar code fragments are called clones. Developers create clones when they reuse code using copy, paste and modify, either within a software system or between software projects, although clones may arise for a variety of other reasons [1]. Studies have shown that 7-23% of a software system is cloned code [2], [3]. By managing or refactoring their clones, developers can improve and maintain software quality, reduce development costs and risks, prevent and detect bugs, and more [1]. Clone management and research studies depend on clone detection tools. In 2013, Rattan et al. [4] found at least 70 tools in the literature, a significant increase from the 40 tools found by Roy et al. [5] in 2009. Despite this interest in new clone detection tools, there has been little evaluation of their performance.

Clone detection tools are typically evaluated using recall and precision. Recall is the ratio of the clones within a software system or repository that a tool is able to detect, while precision is the ratio of the clones reported by a tool that are true clones, not false positives. While time consuming, precision can be measured by validating a random sample of a tool's output. This is typically done, at least informally, by the tool's author during development [1]. On the other hand, recall has been challenging to measure as it requires a benchmark of known reference clones [1].

Clone detection tools should be evaluated using both synthetic and real-world clone benchmarking strategies. Our Mutation and Injection Framework [6], [7] is a synthetic benchmarking technique that uses artificially constructed clones in a mutation-analysis procedure to precisely measure recall at a finer granularity than clone type. While the Mutation Framework can precisely evaluate the capabilities of a tool, it should be complemented with a real-world benchmark. A real-world benchmark evaluates how these capabilities translate into real-world performance for complex clones produced by real developers in real systems. The most widely accepted real-world benchmark is Bellon's Benchmark [8], which Bellon built by manually validating 2% of the clones detected by six contemporary (2002) tools for eight subject systems. We previously showed that this benchmark, and its variants [9], [10], are possibly not suitable for accurately evaluating modern tools [10]. Many of its problems originate from it having been built using tools that are now out of date.

For this reason, we introduced BigCloneBench [11], a real-world benchmark of manually validated clones in the big data IJaDataset-2.0 [12] (25,000 Java systems). It was built, without using clone detectors, by mining IJaDataset for clones of specific functionalities. The current version of the benchmark contains 8 million clone pairs across 43 functionalities. Each clone is semantically similar by its functionality. It contains both intra and inter-project clones spanning the four primary clone types, including the entire range of syntactical similarity.

In this paper, we evaluate the recall of ten clone detection tools using BigCloneBench. We compare these results against our Mutation Framework to better understand the tools, and to evaluate the accuracy of the benchmarks. With BigCloneBench, we examine the differences in intra and inter-project recall. We use multiple clone-matching metrics to evaluate how well the tools capture the benchmark clones with respect to usability in refactoring and clone-analysis use-cases. In summary, we address the following research questions:

- **RQ1** What is the recall of these tools as measured by the real-world benchmark BigCloneBench?
- **RQ2** How does real-world benchmarking compare to synthetic? What do the similarities and differences tell us about tool performance and benchmark accuracy?
- **RQ3** How does intra and inter-project clone recall differ, in particular for the case of an ultra-large dataset?
- **RQ4** What is the clone capture quality of the tools for refactoring and clone analysis use-cases?

978-1-4673-7532-0/15 © 2015 IEEE

II. Related Work

Clone detection recall is essential to understanding the effectiveness of clone detection tools in software development and clone research studies. Accurately measuring recall has been challenging because it requires a large, varied and comprehensive benchmark of reference clones. Benchmarks have been built by manually mining system(s) for clones [11], [13], manually validating a sample of the output of various tools for some subject systems [14], [8], [15], or injecting known clones into a system [6], [7], [10].

The most widely accepted real-world clone benchmark has been Bellon's Benchmark. It is a product of Bellon et al.'s benchmarking experiment [8], which measured the relative recall of six contemporary (2002) tools for four C and four Java subject systems. Bellon created this benchmark by manually validating 2% of the 325,935 candidate clones detected by the tools. Only the true positives, per his judgment, were added to the benchmark, possibly with modifications to their line boundaries. Murakami et al. [9] supplemented the benchmark by identifying the gap lines in the Type-3 reference clones. We contributed improvements to its clone-matching metrics [10].

There are some limitations to Bellon's Benchmark, or any clone benchmark built using the clone detection tools themselves. Since it was built using contemporary (2002) tools, the benchmark is biased and limited by their detection characteristics. The clones these tools were unable to detect form a gap in the benchmark. The effect of this gap has increased as clone detection techniques have improved. Particularly, Type-3 detection has been an area of significant improvement since the benchmark was created. The tools used to construct the benchmark have an advantage because their clones are in the benchmark. This limit's the benchmarks re-usability with modern tools. Adding clones detected by new tools to the benchmark is difficult because Bellon's validation process is not well documented [2], [8], [10]. The benchmark lacks clone variety, as it only considers the clones in four subject systems per language. Additionally, Baker [2] found clone validation and clone-type classification errors in the benchmark.

We introduced a synthetic clone benchmark, the Mutation and Injection Framework [6], [16]. This framework synthesizes customizable benchmarks of artificial copy and paste clones using a mutation-analysis procedure. Clone synthesis is based on an empirically validated taxonomy of the types of edits developers make on copy-pasted code. The framework can measure recall per edit type, a finer granularity than clone type, which enables it to pin point the strengths and weaknesses of a tool. As a synthetic benchmark, it can provide bias-free, controlled and fine granularity benchmarking of a tool's capabilities. We have shown it provides good results with modern tools, and reveals insights into their clone recall [10]. However, benchmarking with real clones is also important. Ideally, both synthetic and real-world benchmarks should be used to gain the advantages of each benchmarking strategy.

In our previous work [10], we evaluated eleven tools using Bellon's Benchmark and our Mutation Framework. We compared the recall measurements of the benchmarks, as well as against our expectations for the tools. The expectations were based on our knowledge and experiences with the tools, as well as feedback from the tool developers, where available. We found anomalies in Bellon's Benchmark's measurement of recall, as well as strong disagreement with the Mutation Framework and expectations. We suggested that Bellon's Benchmark is possibly not appropriate for modern tools.

For this reason, we introduced BigCloneBench [11], a modern real-world clone benchmark of manually validated clones. BigCloneBench was built by mining thousands of software systems for clones of 43 functionalities, ensuring a large variety of intra-project and inter-project clones. All of the clones are semantically similar, and span the entire range of syntactical similarity. Each of the four primary clone types are well represented in the benchmark. It was built without the use of clone detection tools, so it is not biased for or limited to the detection abilities of a particular set of tools.

III. Definitions

Code Fragment: A continuous segment of source code, specified by the triple (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 the triple (f_1, f_2, ϕ), including the similar code fragments f_1 and f_2, and their clone type ϕ.

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

Type-1 (T1): Syntactically identical code fragments, except for differences in white space, layout and comments [1], [8].

Type-2 (T2): Syntactically identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences [1], [8].

Type-3 (T3): Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences [1], [8].

Type-4 (T4): Syntactically dissimilar code fragments that implement the same functionality [1].

Since there is no consensus on the minimum syntactical similarity of a Type-3 clone, it is difficult to separate Type-3 and Type-4 clone pairs that implement the same functionality, as in our BigCloneBench. Instead, we divide the Type-3 and Type-4 clones into four categories based on their syntactical similarity. We define **Very-Strongly Type-3** clones (**VST3**) as those with a similarity in range 90% (inclusive) to 100% (exclusive), **Strongly Type-3** (**ST3**): 70-90%, **Moderately Type-3** (**MT3**): 50-70%, and **Weakly Type-3/Type-4** (**WT3/4**): 0-50% [11]. We measure similarity as the minimum ratio of lines or tokens a code fragment shares with another after Type-1 and Type-2 normalization. Shared lines or tokens are identified by diff [17]. Most tools measure similarity by line or by token. We classify the clones into these categories using the smaller of their line and token-based clone similarity measures.

TABLE I
BIGCLONEBENCH CLONE SUMMARY

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
Number of Clone Pairs	35787	4573	4156	14997	79756	7729291
				7868560		

IV. BIGCLONEBENCH

BigCloneBench [11] is a clone detection benchmark consisting of manually validated clones in IJaDataset 2.0 [12], 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. Functions that might implement a target functionality were identified using keyword and source code pattern heuristics. The identified functions were manually tagged as true or false positives of the target functionality by judges. All true positive functions of a functionality form a large clone class of semantically similar functions. A clone class of size $O(n)$ contains $O(n^2)$ clone pairs. Post-procesing identified the clone types and syntactical similarity of these clones. The reference clones of BigCloneBench are semantically similar, are of the first four clone types, and span the entire range of syntactical similarity. The benchmark contains both intra-project and inter-project clones. Further details about the benchmark may be found elsewhere [11], [18].

The contents of the benchmark are summarized per clone type in Table I. The first version of BigCloneBench targeted 10 functionalities, and was released [18] alongside its publication [11]. We use a snapshot of the in-progress expansion of the benchmark, which contains clones of 43 functionalities. We use only the clones that are at least 6 lines and 50 tokens in length. This allows us to configure the tools appropriately for clone size. This is a typical minimum clone size used by tools [1] and previous benchmark experiments [8]. We removed the source files that are 2000 lines in length or longer and their clones. These files make up an insignificant portion of IJaDataset (6238 files), but significantly impact the execution requirements (time, memory) of the clone detectors.

V. THE MUTATION AND INJECTION FRAMEWORK

Our Mutation and Injection Framework [6] is a synthetic benchmark for evaluating clone detection tools. It uses source-code mutation and injection to generate a large and customizable benchmark of artificial clones. Mutation analysis is a well studied and accepted approach for evaluating software testing quality [19], and we use a similar methodology to evaluate clone detection tools. A code fragment is extracted from a large repository of varied source code. It is duplicated and mutated by a mutation operator that introduces a single random code edit. The framework uses fifteen mutation operators that are based on a comprehensive and empirically validated taxonomy of the types of edits developers make on copy and pasted code [5]. The mutation operators are shown in Table II, and span the first three clone types. The original and mutant code fragments are injected into a copy of a subject system, evolving the system by a single copy, paste and modify clone. This is repeated thousands of times to build a large corpus of mutant systems. The subject tools are then executed for these mutant systems, and their recall is measured specifically for the injected clones. The framework is fully automated. Further details are in our previous publications [6], [16], [7].

TABLE II
CLONING MUTATION OPERATORS FROM CLONE EDITING TAXONOMY

ID	Mutation (Edit) Description	Clone Type
mCW_A	Addition of whitespace.	Type-1
mCW_R	Removal of whitespace.	
mCC_BT	Change in between token (/* */) commenting.	
mCC_BT	Change in end of line (//) commenting.	
mCF_A	Change in formatting (add newline).	
mCF_R	Change in formatting (remove newline).	
mSRI	Systematic renaming of an identifier.	Type-2
mARI	Renaming of a single identifier instance.	
mRL_N	Change in value of a numeric literal.	
mRL_S	Change in value of a string literal.	
mSIL	Small insertion within a line.	Type-3
mSDL	Small deletion within a line.	
mIL	Insertion of a line.	
mDL	Deletion of a line.	
mML	Modification of a line.	

VI. EXPERIMENT

Big Clone Bench. We executed the tools for IJaDataset 2.0, and measured their recall for the clones in BigCloneBench. These subject tools were generally designed for clone detection within a single software system, or a small collection of software systems. None of these tools can scale to IJaDataset on ordinary hardware. Our goal is to measure recall using a large number of clones, not to evaluate the scalability of the tools. We avoid the scalability issue by executing the tools for smaller subsets of IJaDataset that expose the tools to every clone in BigCloneBench. We executed the tools for one subset per functionality in BigCloneBench. Each subset includes every file that contains a function judged as a true or false positive of the subset's functionality during the mining process. Therefore, each subset contains a mix of true and false clones. If a subset was too large for a tool to evaluate within memory constraints (12GB), we partitioned the subset into smaller sets and executed the tool for each pair of partitions. A tool's clone detection reports were merged before evaluation.

With BigCloneBench, we use a coverage-based clone matching metric to determine if a reference clone in the benchmark is successfully detected by a candidate clone reported by a tool. The **coverage-match**, or *c-match* for short, is based on our *covers* metric. A code fragment f_1 covers code fragment f_2 if it intersects a ratio t of the source lines of f_2, as shown in (1), given that the code fragments are in the same source file. A candidate clone, C, matches a reference clone, R, by the *c-match* if its code fragments cover a ratio t of the reference clone's code fragments, as shown in (2). When the metric is evaluated, both orderings of the candidate clone's code fragments are tested. We configured

the metric with a 70% minimum coverage threshold. This is a conservative threshold, neither too strict nor too generous, that has been used in previous benchmarking experiments [8], [10]. A tool's recall is therefore the ratio of the reference clones in the benchmark that are matched by candidate clones reported by the tool, as judged by the *c-match* clone-matching metric.

$$covers(f_1, f_2, t) = \frac{min(f_1.e, f_2.e) - max(f_1.s, f_2.s) + 1}{f_2.e - f_2.s + 1} \geq t \quad (1)$$

$$c\text{–}match(C, R, t) = covers(C.f_1, R.f_1, t) \wedge covers(C.f_2, R.f_2, t) \quad (2)$$

Mutation and Injection Framework. We set the framework to randomly extract 250 functions from a source repository and, from each, create 15 mutant functions using the 15 mutation operators (3,750 clone pairs). Each clone was randomly injected into 10 unique copies of a subject system (37,500 mutant systems). We used IPScanner as our subject system, and JDK6 and Apache Commons as our source repository. We constrained the benchmark to the following clone properties: (1) 15-200 lines in length, (2) 100-2000 tokens in length, and (3) mutations do not occur within the first and last 15% of a code fragment by line. The 15% mutation containment ensures that the introduced edits occur within the clone, and not on its edges. Clone detection time often scales with minimum clone size, so we used a larger minimum clone size to make execution of the tools for 37,500 systems practical. We measured the syntactical similarity of the Type-3 clones and found that they correspond to the Very-Strongly Type-3 similarity region. These are different clones than we used in our comparison with Bellon's Benchmark [10]. Here we generated function clones for best comparison with BigCloneBench, whereas we previously generated block clones for best comparison with Bellon's Benchmark.

Recall is measured by a subsume-based clone-matching metric that is parameterized with the mutation containment. For a mutation containment of 15%, the metric considers a candidate clone to subsume a reference clone even if the candidate misses the first and/or last 15% of the reference clone's code fragments. This is essentially the *c-match* metric with the added restriction that the candidate must cover the inner 70% of the reference. This restriction ensures that any candidate accepted as a match of a reference clone has captured the clone-type specific edits added to the reference by a mutation operator. Therefore, recall measured by the Mutation Framework reflects a tool's ability to handle the specific clone edit types from the taxonomy (Table II).

Tool Configuration. The subject tools, the clone types they can detect, and their configurations for the benchmarks, are summarized in Table III. We wanted the recall measurements to reflect what an experienced user can expect with their own systems. An experienced user has explored a tool's parameters and documentation, and modifies the default settings for their use-case. We configured the tools from a user-perspective by considering: (1) the default settings, (2) the documentation, and (3) the known properties of the target benchmark, which include clone types, syntactical similarity, and clone size. We also consulted the tool developers, where available. We enable any supported Type-1 and Type-2 normalizations. We configured the tools with Type-3 sensitivity thresholds based on their defaults and documentation. While greatly lowering their syntactic similarity threshold may enable them to detect more clones in BigCloneBench [20], which contains clones across the entire spectrum of syntactical similarity, their lack of semantic awareness would also cause them to detect a large number of false positives. Users are most likely to follow the recommended thresholds, so our results reflect standard usage of the tools. If a setting was not well documented, we experimented with it to observe its effect. We avoided over-configuring or over-optimizing the tools for the benchmark, as a user would not be able to do this for their own systems. We also avoided configuring the tools in a way that would increase recall at the expense of precision. The per benchmark configurations are mostly the same, except for differences in minimum clone size. Both benchmarks have a strict minimum clone size, so the tools could be configured for clone size with confidence. While different configurations may improve recall, these configurations reflect usage by an experienced user.

TABLE III
SUBJECT TOOLS AND CONFIGURATIONS

Tool	Types	BigCloneBench	Mutation Framework
CCFinderX [21]	1,2	Min length 50 tokens, min token types 12.	Min length 50 tokens, min token types 12.
ConQat [22]	1,2,3	Min length 6 lines, max errors 5, gap ratio 30%.	Min length 15 lines, max errors 3, gap ratio 30%.
CPD [23]	1,2	Min length 50 tokens, ignore annotations/identifiers/literals, skip parser errors.	Min length 100 tokens, ignore annotations/identifiers/literals, skip parser errors.
CtCompare [24]	1,2	Min length 50 tokens, max 6 isomorphic relations.	Min length 100 tokens, max 3 isomorphic relations.
Deckard [25]	1,2,3	Min length 50 tokens, 85% similarity, 2 token stride.	Min length 100 tokens, 85% similarity, 4 token stride.
Duplo [26]	1	Min length 6 lines. Min 1 character per line.	Min length 15 lines. Min 1 character per line.
iClones [27]	1,2,3	Min length 50 tokens, min block 20 tokens.	Min length 100 tokens, min block 20 tokens.
NiCad [28]	1,2,3	Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity.	Min length 15 lines, blind identifier normalization, identifier abstraction, min 70% similarity.
SimCad [29]	1,2,3	Greedy transformation, unicode support, min 6 lines.	Greedy transformation, unicode support, min 15 lines.
Simian [30]	1,2	Min length 6 lines, ignore identifiers and literals.	Min length 15 lines, ignore identifiers and literals.

VII. BENCHMARK RESULTS

In this section, we measure the recall of the tools using BigCloneBench (**RQ1**) and the Mutation Framework. We compare these results, and interpret what the similarities and differences between the benchmarks tell us about the tool performance and benchmark accuracy (**RQ2**). The recall measurements by the benchmarks, and their differences, are show in Table IV. Due to space considerations, we do not show Mutation Framework recall per mutation operator. Instead, we summarize recall per clone type by averaging across the mutation operators that produce a particular clone type (Table II). The Mutation Framework's Type-3 clones best match the syntactical similarity of the Very-Strongly Type-3 clones in BigCloneBench, so we compare using this Type-3 category.

TABLE IV
BENCHMARK RECALL MEASUREMENTS AND DIFFERENCE PER CLONE TYPE

Tool	BigCloneBench						Mutation Framework			Difference		
	T1	T2	VST3	ST3	MT3	WT3/T4	T1	T2	VST3	ΔT1	ΔT2	ΔVST3
CCFinderX	100	93	62	15	1	0	99	70	0	1	23	62
ConQat	67	90	73	33	1	0	91	90	86	-24	0	-13
CPD	100	94	71	21	1	0	99	82	0	1	12	71
CtCompare	95	78	59	17	0	0	96	63	0	-1	15	59
Deckard	60	58	62	31	12	1	39	39	37	21	19	25
Duplo	89	74	46	8	0	0	38	0	0	51	74	46
iClones	100	82	82	24	0	0	100	92	96	0	-10	-14
NiCad	100	100	100	95	1	0	100	100	100	0	0	0
SimCad	100	98	91	48	8	0	100	94	89	0	4	2
Simian	95	78	53	13	0	0	81	90	0	14	-12	53

TABLE V
BIGCLONEBENCH: TYPE-3 RECALL

Tool	Syntactical Similarity Interval, x% to x+5%									
	50	55	60	65	70	75	80	85	90	95
ConQat	0	0	2	3	8	18	41	62	85	56
Deckard	10	14	15	18	24	28	39	34	52	75
iClones	0	0	1	2	5	19	36	39	75	91
NiCad	0	0	1	10	85	99	100	100	100	100
SimCad	5	7	13	16	23	45	46	77	85	99
CCFinderX	0	2	2	1	5	8	23	25	51	77
CPD	0	0	2	2	5	20	22	35	73	68
CtCompare	0	0	1	2	4	19	15	29	62	54
Duplo	0	0	0	0	1	3	7	17	36	60
Simian	0	1	0	1	2	5	22	23	45	63

A. BigCloneBench

Type-1. CCFinderX, CPD, iClones, NiCad and SimCad have perfect Type-1 recall. CtCompare and Simian also have excellent recall at 95%. Duplo has good recall at 89%. Conqat (67%) and Deckard (60%) fall behind the others.

Type-2. Only NiCad has perfect recall for the Type-2 clones. CCFinderX, ConQat, CPD and SimCad have excellent recall, all $\geq 90\%$. iClones maintains good recall at 82%. CtCompare, Duplo and Simian have decent recall in the 70%s, while Deckard has poor recall at 58%. Duplo performs well despite not supporting Type-2 normalizations. These Type-2 clones must contain a significant Type-1 region that Duplo detects and which is accepted by the *c-match*'s 70% coverage requirement.

Type-2 detection reduces to Type-1 detection after Type-2 normalizations are applied. Where Type-2 recall is lower than Type-1, we expect the tool is missing or struggling with particular Type-2 normalization(s). It is strange that ConQat has a very strong Type-2 recall, but the weakest Type-1 recall of these tools. Specifically, ConQat is not detecting Type-1 clones of a single particular functionality which contributed a large number of Type-1 clones to the benchmark. We were unable to determine why ConQat was missing these clones. Re-configuring ConQat for them made no difference. Ignoring these clones, ConQat has a Type-1 recall of 97%.

Type-3. For the Very-Strongly Type-3 clones, NiCad has perfect recall and SimCad has excellent recall (91%). iClones has good detection (82%), while ConQat (73%) and CPD (71%) have decent recall. The remaining tools have poor recall for these clones. NiCad has excellent (95%) detection for the Strongly Type-3 clones, while the other tools have poor detection. None perform well in the Moderately Type-3 or Weakly Type-3/Type-4 regions. Semantic-awareness may be needed to detect clones in these regions with good precision.

Many of the tools that do not formally support Type-3 clone detection (Tabel III) have recall in the Very-Strongly Type-3 similarity region. In particular, CCFinderX and CPD have similar recall to the Type-3 detectors Deckard and ConQat. This is due to these tools detecting significant continuous Type-1 or Type-2 regions that cover at least 70% of a Type-3 reference clone. It is more desirable for a tool to include the Type-3 regions in its detection of these clones. Otherwise, the user has to manually recognize the larger Type-3 clone.

The Type-3 detectors have lower Type-3 recall than we expected. We therefore investigate Type-3 recall for finer grained syntactical similarity regions. Table V shows Type-3 recall per 5% interval of syntactical similarity. For example, the 75% interval includes all Type-3 clones with similarity in the range 75% (inclusive) to 80% (exclusive). We do not show recall below 50% similarity as none of the tools have noteworthy recall in that range. We split the tools in this table based on formal Type-3 detection support. Only the tools above the splitting line feature Type-3 detection.

ConQat has good recall (85%) for Type-3 clones in the 90% interval. Oddly, it has significantly poorer recall for the more similar clones in the 95% interval, only 56%. It also has poor (62%) recall for the 85% interval. Its recall drops as expected for lower intervals. ConQat was configured with a 70% similarity threshold, and a maximum of 5 errors (Type-3 gaps). The error setting may be holding back ConQat's Type-3 detection. This setting has a strong impact on execution time, and we are already using a value larger than default (3). Deckard does not have high recall for any of the intervals, with only decent (75%) recall for the 95% interval. It has poor recall for the 85% and 90% intervals, despite being configured with its default 85% similarity threshold. iClones has excellent (91%) recall for the 95% interval, but only decent (75%) recall for the 90% interval, with very low recall for the lower intervals. iClones does not present a setting to increase Type-3 sensitivity. NiCad has perfect recall for Type-3 clones with at least 80% similarity. It has excellent recall for the 75% interval, and good recall for the 70% interval. Its recall drops sharply for intervals below 70%, which is expected as it was configured for a 70% similarity threshold. SimCad has excellent (99%) recall for the 95% interval, and good (85%) recall for the 90% interval. Its recall drops below 50% for the 80% interval and below. SimCad's SimHash sensitivity threshold was chosen empirically by the tool authors, so modification is not recommended. Deckard and SimCad are the only tools to have a notable, although small, recall for similarity intervals below 65%. Their detection strategies (AST, SimHash) may be more resilient to statement re-ordering.

Of the tools lacking formal Type-3 support, CCFinderX and CPD have the best recall for Type-3 clones, specifically for clones in the 90% and 95% intervals. Tools lacking Type-3 detection are able to detect Type-3 clones, by the *c-match*,

when the clones contain a continuous Type-1 or Type-2 region covering at least 70% of the reference clone. These tools have poor Type-3 recall below 90% similarity, showing this scenario becomes rare for lower syntactical similarity.

Many of these tools have very strong Type-1 and Type-2 recall. Many perform well for the Very-Strongly Type-3 clones. Only NiCad performs well for the Strongly Type-3 clones. ConQat, Deckard and NiCad have Type-3 sensitivity configurations, which we left at their default values. Presumably these defaults were selected by the tool authors considering precision. The default values are what tool users are most likely to use, so these results reflect typical tool usage. The tools could be improved with increased Type-3 sensitivity, although this must be done while maintaining precision.

B. Mutation and Injection Framework

Type-1. Most of the tools have excellent Type-1 recall ($\geq$90%), with perfect detection by iClones, NiCad and SimCad. The exception is Deckard and Duplo which have poor recall. Simian has good recall, but falls behind the other tools.

Type-2. ConQat, iClones, NiCad, SimCad and Simian have excellent recall, $\geq$90%. CPD has good recall, CCFinderX and CtCompare have decent recall, while Deckard again has poor recall. The Mutation Framework correctly identifies that Duplo does not support Type-2 normalizations and detection.

Type-3. The Mutation Framework correctly identifies that CCFinderX, CPD, CtCompare, Duplo and Simian do not support Type-3 detection. While they may be able to detect Type-1 or Type-2 regions within the Type-3 references, they are unable to capture the Type-3 regions. The Mutation Framework requires the tool to capture the clone-type specific edit it introduced to the synthesized clones. Most of the Type-3 tools perform well. iClones and NiCad have excellent recall for the Type-3 clones, both >95%. ConQat and SimCad also perform well, with >85%. Only Deckard performs poorly, with only 37% recall. Deckard performs uniformly poor across the clone types, suggesting that its weakness is not due to the handling of any particular clone-type specific difference. We believe this is due to it's outdated Java parser (Java-1.4 only).

C. Comparing the Benchmarks

Here we compare the BigCloneBench and Mutation Framework results for these tools (**RQ2**). These benchmarks use very different, but complementary, benchmarking strategies. The advantage of the Mutation Framework's synthetic benchmarking technique is it measures recall per clone type very precisely. Since it synthesizes its reference clones, it is able to determine if a tool successfully detected the clone-type specific regions of a reference clone. For example, the Mutation Framework will not accept a candidate clone as a match of a Type-3 reference clone if the candidate clone does not capture the Type-3 regions. Additionally, each reference clone contains only one type-specific change from the taxonomy (Table II), so clone-type-specific recall can be measured without bias due to features from the other clone types. The advantage of BigCloneBench's real-world benchmarking technique is it measures recall using real clones in real systems. The distribution of the clone types and features of the reference clones reflects what is found in real systems. BigCloneBench has complex clones that contain mixed features of the clone types. While the Mutation Framework can measure per clone-type recall more precisely, BigCloneBench shows how the tools perform for real clones. The advantages of both of these benchmarks are essential for understanding clone detection recall. We suggest that both benchmarking strategies are needed to fully evaluate the tools.

Since the benchmarks use very different methodologies, we consider them to agree if their recall measurement has an absolute difference no greater than 15%. This is the threshold we have used in previous work when comparing benchmarks [10]. We have highlighted in gray the cases where the frameworks disagree. The cases highlighted with light gray are cases where we expected disagreement due to the Mutation Framework's precise clone-type recall measurements. The cases highlighted with dark gray are the cases we did not anticipate.

The light gray highlighted cases are where the Mutation Framework has measured no recall, while BigCloneBench has measured a significant (>15%) recall. Neither benchmark is incorrect in these cases. Rather, each benchmark is telling us something different about the tools, as per their individual benchmarking advantages. These are cases where a tool does not formally support a clone type. For example, the light gray highlighted tools under the 'ΔVST3' header do not formally support Type-3 detection. The Mutation Framework requires the tools to detect the type-specific changes in the reference clones, so it measures no Type-3 recall for these tools. However, these tools may detect a significant ($\geq$70%) Type-1 or Type-2 region in the Type-3 clones, so BigCloneBench measures a sizable recall. BigCloneBench tells us these tools can detect significant portions of the Type-3 clones, but the Mutation Framework tells us they cannot detect the portions containing the Type-3 differences. The quality of these tools' Type-3 detection is therefore very limited. They are not appropriate for automatic clone analysis, as automated tools will see these clones as Type-1 or Type-2, and provide incorrect analysis. These tools may also be inconvenient for use-cases involving manual inspection, as users will need to manually recognize the Type-3 features missed by the tool. These conclusions also hold for Duplo, which does not formally support Type-2 detection.

The cases highlighted with dark gray were not expected. The benchmarks disagree for Deckard for all clone types, with the Mutation Framwork uniformly measuring lower recall. We believe this is due to limitations in Deckard's parser, which only supports the Java-1.4 specification. The Mutation Framework synthesized clones using code fragments from JDK6 and Apache Commons, both of which make significant use of generics (Java-1.5). Deckard may perform better for BigCloneBench if Java-1.5+ features are less commonly used. The Mutation Framework measures a significantly lower Type-2 recall for CCFinderX. We investigated the per mutation-operator recall of CCFinderX, and observed it only has low

recall for Type-2 clones where a single instance of an identifier is renamed, but has good recall for the other Type-2 edit types (Table II). Perhaps this edit-type is rarer than the others in real-world clones, which is why BigCloneBench measures a higher recall. ConQat has significantly lower Type-1 recall measured by BigCloneBench. This is its poor detection of Type-1 clones from a particular functionality, as we mentioned earlier. We are not sure why Duplo's Type-1 recall is significantly lower with the Mutation Framework.

Overall, the benchmarks agree for most recall measurements. Ignoring the cases where the tools do not formally support a clone type (light-gray highlight), the benchmarks agree in 70% of the Type-1 cases, 78% of Type-2 cases, and 80% of Type-3 cases. Half of the cases of disagreement that are not related to clone type support are from Deckard, which is likely due to its parser limitations. This strong agreement between two very different benchmarking strategies builds our confidence that the measurements are accurate (**RQ2**).

VIII. Intra-Project vs. Inter-Project Performance

Traditionally, clone detectors have been designed to locate clones within a single software system. However, applications of clone detection extend to clones between distinct software systems. Intra and inter-project clones may have different properties, so the tools may have different recall for these contexts. In this section, we compare the intra and inter-project recall of the tools using BigCloneBench (**RQ3**). For intra-project recall, we evaluate the tools only for the reference clone pairs whose code fragments are located in the same software system. This is the average recall of the tools in a traditional single-system clone detection scenario. For example, when a developer uses a clone detector to locate the clones in their software project. For inter-project recall, we consider only the reference clone pairs whose code fragments are located in different software systems. This is the average recall of the tools in a cross-project clone detection scenario. For example, when a company uses clone detection to locate code duplication across their products, or when a researcher studies code duplication across the open-source community.

Table VI summarizes intra and inter-project recall per clone type, as well as their absolute difference. We do not include the MT3 and WT3/T4 categories as the tools have negligible recall for these clone categories. We consider a tool's difference in recall to be significant if it exceeds 15%, and these cases are highlighted in gray. We choose this threshold based on the distribution of the difference across these 40 per tool, per clone-type, cases. The average difference is $\pm 13\%$. The average is pulled up by a handful of cases with considerable difference. Only 15 of the cases have a difference that meets or exceeds the average. These 15 cases have an average difference of $\pm 28\%$, while the other 25 cases have a average difference of $\pm 4\%$. We use the average difference of the 40 cases, rounded up to 15%, as our threshold. We believe we are being sufficiently cautious with this threshold, and are confident the cases exceeding the threshold are affected by differing properties of intra and inter-project clones.

Most of these tools exhibit significant differences between their intra-project and inter-project recall for at least one of the clone types. Only NiCad exhibits no significant differences between these clone contexts. ConQat has significantly different recall for four of the clone types. Generally, it has better recall for inter-project clones, although it has better intra-project recall for the ST3 clones. Duplo has considerably better intra-project recall for Type-1 clones, yet considerably better inter-project recall for Type-2 clones. CCFinderX, CPD, CtCompare, Deckard, iClones, SimCad and Simian have a difference in recall for only one of the clone types.

Many of the tools have significant differences in Type-2 recall, with better inter-project recall in each of these cases. This difference is considerable, with a -35% difference on average. Difference in Type-1 recall is not uniform, although the difference is considerable for ConQat and Duplo. Similarly for the VST3 recall, with ConQat and Deckard showing a considerable difference. The differences in ST3 recall are not as strong as for the other types. Perhaps intra and inter-project Type-3 clones in this similarity range have similar properties, or the tools are generally not sensitive to their differences.

While the participating tools were primarily designed for single-system clone detection, our findings show that they do not have a universal weakness in cross-project clone detection. Per clone type, some of the tools perform better for inter-project clones, some for intra-project clones, and in most cases no significant difference is found. Of the thirteen cases of significant difference, seven cases prefer inter-project recall, while six prefer intra-project recall. Five of these cases show significantly better inter-project recall for Type-2 clones, often by a considerable amount. These results can be used by users to decide which tool is best for their use-case.

IX. Clone Capture Quality

While high recall is important, it is also important that a tool capture the clones in a way that is useful to a user's clone-related task (**RQ4**). We evaluated the recall of these tools using BigCloneBench and our coverage clone-matching metric (*c-match*). This metric accepts a candidate clone that covers 70% of a reference clone's source lines. This metric ignores any additional lines the tool reports beyond the boundaries of the reference clone. Since the reference clones of BigCloneBench are function clones, additional lines reported by the tool are external to the functions. These source lines may be from other functions surrounding the function clone, or class-definition syntax. Ideally, clone detection tools should respect function boundaries when reporting clones. Tools that report clones that extend beyond function boundaries have poorer clone capture quality because these clones have poorer usability.

Some primary use-cases of clone detection include refactoring, clone management and automatic clone analysis. Clones that extend beyond function boundaries, or that intersect multiple functions, do not imply any specific refactoring action [1], [8]. This requires the developer to manually trim and/or split the clone by functional boundaries before considering any refactoring tasks. In clone management, developers need to

TABLE VI
BIGCLONEBENCH: INTRA-PROJECT VS INTER-PROJECT RECALL

Tool	Intra-Project Recall				Inter-Project Recall				Difference, $\Delta = (Intra - Inter)$			
	T1	T2	VST3	ST3	T1	T2	VST3	ST3	ΔT1	ΔT2	ΔVST3	ΔST3
CCFinderX	100	89	70	10	98	94	53	17	2	-5	17	-7
ConQat	62	60	57	49	98	95	91	25	-36	-35	-34	24
CPD	100	80	67	18	100	96	76	22	0	-16	-9	-4
CtCompare	96	38	52	14	88	85	66	19	8	-47	-14	-5
Deckard	59	60	76	31	64	58	46	30	-5	2	30	1
Duplo	97	34	49	12	50	81	42	6	47	-47	7	6
iClones	100	57	84	33	100	86	78	20	0	-29	6	13
NiCad	100	100	100	99	100	100	100	93	0	0	0	6
SimCad	100	95	86	59	100	99	96	43	0	-4	-10	16
Simian	98	82	55	6	77	77	50	16	21	5	5	-10

reason about a large number of clones, and the appropriate actions to prevent harm to software quality. Having to manually trim or split individual clones significantly increases the difficulty and cost of reasoning about a large number of clones.

Automatic clone analysis is used by development tools that aid clone refactoring and management, as well as by researchers who study software using clones. An example of automatic clone analysis is a development tool that monitors the changes a developer makes to a function, and recommends clones detected by a tool that the developer most likely wants to propagate the changes to. A clone analyzer reasons about clones for the developer or researcher when there are too many clones for them to manually investigate. However, the analyzer may behave incorrectly, or produce poor results, when the clone spans multiple functions. The analysis algorithm and metrics likely assume that the input clones are contained within a single logical unit of code (e.g. function, block). In general, it is not useful for a tool to report a clone that extends beyond the boundaries of a function. Clones that span multiple functions are not meaningful since the order and position of functions in a class is not meaningful.

While respecting function boundaries is only one consideration of clone capture (i.e., reporting) quality, we have shown why it is important to the practical usability of the clones. In this section, we evaluate how well the tools respect function boundaries in their detection of the reference clones. We do this using two extensions of our *c-match* metric.

The **strict coverage metric**, *sc-match*, extends the *c-match* to also require the candidate clone to not extend more than l lines beyond the reference function clone's boundaries, as shown in (3). For this evaluation, we use a tolerance of 3 lines. This is a small enough extension beyond the boundaries of a clone that even automatic analysis could trim the candidate clone to the function boundaries without having to split the additional lines into an independent clone. It is small enough that a user should require minimal effort to visually recognize and ignore the extraneous lines past the function boundary.

$$\begin{aligned} sc\text{–}match(C,R,t,l) = c\text{–}match(C,R,t) \wedge \\ C.f_1.s \geq R.f_1.s - l \ \wedge C.f_2.s \geq R.f_2.s - l \wedge \\ C.f_1.e \leq R.f_1.e + l \ \wedge C.f_2.e \leq R.f_2.e + l \end{aligned} \quad (3)$$

Some tools may not respect function boundaries when reporting clones. Such a tool is a poor choice for automatic refactoring and analysis usages. However, as long as the target reference clone is clearly featured in the reported candidate clone, a user should be able to visually parse the reference clone with an acceptable increase in effort. To evaluate the tools from this perspective, we use the **featured coverage metric**, or *fc-match*. This metric requires the candidate clone to cover the reference clone, and for the covered portion of the reference clone to be a significant feature of the candidate clone. This is the *c-match* in both directions, as shown in (4). We decided a 70% coverage of the candidate clone is the minimum for the reference clone to be visually identifiable by the user without a significantly burdensome examination.

$$fc\text{–}match(C,R,t) = c\text{–}match(C,R,t) \wedge c\text{–}match(R,C,t) \quad (4)$$

If recall measured by the *sc-match* and the *c-match* is similar, then we know that the tool respects function boundaries when reporting the reference clones, and is therefore a good candidate for both manual and automatic refactoring and analysis use-cases. If recall by the *sc-match* is much lower than by the *c-match*, then the tool does not respect function boundaries. Such a tool is not appropriate for use-cases that use automatic analysis. However, if the tool has similar recall measured by the *fc-match* and *c-match*, than the tool features the function reference clones in its detection of them. Such a tool is appropriate for use-cases that use manual analysis, although with some increased effort compared to a tool that respects function boundaries. A tool with significantly lower *sc-match* and *fc-match* than *c-match* recall neither respects function boundaries nor features the reference clones. Such a tool is likely burdensome to use for most use-cases.

We compare the tools' recall per clone type for the *c-match*, *sc-match* and *f-match* in Table VII. We include only the VST3 and ST3 Type-3 categories as the tools do not have appreciable recall for the other Type-3/4 categories. We observe trends in these results related to the clone types the tools support, so we organize the tools with those that formally support Type-3 detection above the splitting line. We consider the *sc-match* or *fc-match* recall to be similar to the *c-match* recall if the relative difference is no greater than 20%. We use a relative difference because the tools all have different base recall performances. We use a generous threshold to favor the tools in this evaluation. We highlight in gray these cases of similarity. A highlighted *sc-match* recall indicates the tool respects function boundaries for that clone type, while

TABLE VII
BIGCLONEBENCH: CLONE CAPTURE QUALITY - METRIC COMPARISON

Tool	T1			T2			VST3			ST3		
	C	SC	FC	C	SC	FC	C	SC	FC	C	SC	FC
ConQat	67	67	67	90	90	90	73	73	73	33	33	33
Deckard	60	59	59	58	58	58	62	57	57	31	25	26
iClones	100	100	100	82	82	82	82	82	82	24	24	24
NiCad	100	100	100	100	100	100	100	100	100	95	95	95
SimCad	100	100	100	98	98	98	91	91	91	48	48	48
CCFinderX	100	8	15	93	10	72	62	33	51	15	9	10
CPD	100	14	21	94	19	20	71	38	55	21	17	17
CtCompare	95	1	3	78	3	4	59	25	27	17	15	15
Duplo	89	1	5	74	70	72	46	26	26	8	7	7
Simian	95	12	19	78	51	52	53	25	47	13	11	11

a highlighted *fc-match* indicates the tool features clones of that type when it detects them. The threshold indicates these conclusions hold for at least 80% of the reference clones the tool successfully detects by the *c-match*.

The results show that the Type-3 clone detection tools have exceptional respect for function boundaries. Except for Deckard, recall measured by the *sc-match* and *fc-match* is identical to that measured by the *c-match* for these tools. Deckard has only minor reduction in recall between the *c-match* and *sc-match*. In our experiences with Deckard, it occasionally has errors in its clone boundaries. However, the effect seems to be minimal with respect to the reference clones. The Type-3 clone detectors (ConQat, Deckard, iClones, NiCad, SimCad) have excellent clone capture quality with respect to function boundaries, and are good candidates for any manual and automatic clone analysis use-cases (**RQ4**).

Conversely, for most of the clone types, the tools lacking formal Type-3 detection capabilities do not respect function boundaries, nor strongly feature their detection of the reference clones. In most cases, particularly for the Type-1 and Type-2 clones, their recall by the *sc-match* and *fc-match* is significantly lower than by the *c-match*. The exception is for the Strongly Type-3 clones, and a couple other instances. These tools do not support Type-3 detection. Their recall for Type-3 clones is due to the detection of a significant (70%) Type-1 or Type-2 region within the Type-3 clones. Most of these tools are respecting function boundaries for their detection of the ST3 clones. This is not because these tools respect function boundaries in general, but because the gaps in these clones is bounding the Type-1 or Type-2 region detected by these tools to within the function boundaries. This is lost with the VST3, where there are fewer gaps, and therefore it is less likely the the gaps will bound these tools detection within the function boundaries. Similarly for Duplo, which does not formally support Type-2 detection. Outliers are CCFinderX and Simian, that feature only the VST3 reference clones in their detection. These results show that the tools lacking Type-3 detection (CCFinderX, CPD, CtCompare, Duplo, Simian) neither respect function boundaries nor feature the function clones in their detection of them. Therefore, these tools are not appropriate for automatic analysis, and they may be burdensome for use-cases with manual inspection (**RQ4**).

X. THREATS TO VALIDITY

Alternate configurations of the tools may result in better or worse recall. Wang et al. [31] refer to this as the confounding configuration choice problem, and it is a challenge in all clone studies. We took steps to ensure the tool configurations were appropriate for our study. We used configurations that target the known properties of the benchmark, such as clone types and clone size. Otherwise, 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 [20]. However, the tools would have poor precision for low similarity thresholds.

XI. CONCLUSION AND FUTURE WORK

Recall is an important measure for understanding the effectiveness of clone detection tools in software development and research studies. It has been challenging to measure as it requires a varied and comprehensive benchmark of known clones. Bellon's Benchmark [8] is an influential clone benchmark in the research community, however its dependence on clones detected by contemporary (2002) tools means it may not be suitable for modern tools [10]. Our Mutation Framework [6] synthesizes a benchmark of clones, and can precisely measure recall at a granularity lower than clone-type. However, it is also important to measure recall using real clones produced by developers. We introduced BigCloneBench [11] as a big data, varied and comprehensive clone benchmark for modern tools. In this study, we evaluated ten clone detection tools using BigCloneBench (**RQ1**), and compared these results against our Mutation Framework (**RQ2**). We found the tools have strong detection of Type-1 and Type-2 clones, as well as Type-3 clones with high syntactical similarity. Improvement is needed in the detection of Type-3 clones with lower syntactical similarity, as well as Type-4 clones, while maintaining high precision, which may require semantic awareness. These real-world and synthetic benchmarks have high agreement, so we are confident in their accuracy (**RQ2**). Since BigCloneBench contains both intra and inter-project clones, we were able to evaluate the tools for these contexts. We found that while many of the tools have different recall for single-system and cross-project detection scenarios, neither context was universally favored by the tools (**RQ3**). Using multiple clone-matching metrics with BigCloneBench, we showed that only the Type-3 tools respect function boundaries when reporting clones (**RQ4**). Clones reported by the other tools may have poorer usability in refactoring and automatic clone analysis use-cases. With BigCloneBench and the Mutation Framework, we believe we have created a solid foundation for measuring the recall of clone detection tools.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing, Queens University, Tech. Rep. TR 2007-541, 2007, 115 pp.

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

[3] C. Roy and J. Cordy, "An empirical study of function clones in open source software," in *WCRE '08*, Oct 2008, pp. 81–90.

[4] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165 – 1199, 2013.

[5] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. of Comput. Program.*, pp. 577–591, 2009.

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

[7] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *ICST'09 Mutation Workshop*, 2009, pp. 157–166.

[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] H. Murakami, Y. Higo, and S. Kusumoto, "A dataset of clone references with gaps," in *MSR'14*, 2014, pp. 412–415.

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

[11] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *ICSME*, 2014, p. 5.

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

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

[14] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *SCAM*, 2002, pp. 36–43.

[15] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

[16] C. K. Roy and J. R. Cordy, "Towards a mutation-based automatic framework for evaluating code clone detection tools," in *C3S2E '08*. New York, NY, USA: ACM, 2008, pp. 137–140.

[17] P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower, "Diffutils - gnu project - free software foundation," http://www.gnu.org/software/diffutils, 2015.

[18] "BigCloneBench Downloads," https://github.com/clonebench/BigCloneBench, April 2015.

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[20] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, March 2015, pp. 201–210.

[21] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *Softw. Eng., IEEE Trans. on*, vol. 28, no. 7, pp. 654–670, 2002.

[22] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective - a workbench for clone detection research," in *ICSE*, 2009, pp. 603–606.

[23] "Cpd," http://pmd.sourceforge.net/.

[24] W. Toomey, "Ctcompare: Code clone detection using hashed token sequences," in *IWSC*, 2012, pp. 92–93.

[25] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.

[26] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *ICSM*, 1999, pp. 109–118.

[27] N. Göde and R. Koschke, "Incremental clone detection," in *CSMR*, 2009, pp. 219–228.

[28] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, 2008, pp. 172–181.

[29] M. Uddin, C. K. Roy, and K. A. Schneider, "Simcad: An extensible and faster clone detection tool for large scale software systems," in *ICPC*, 2013, pp. 236–238.

[30] "Simian," http://www.harukizaemon.com/simian/.

[31] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *ESEC/FSE*, 2013, pp. 455–465.