The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis

Jeffrey Svajlenko Chanchal k. Roy

Abstract—An abundant number of clone detection tools have been proposed in the literature due to the many applications and benefits of clone detection. However, there has been difficulty in the performance evaluation and comparison of these clone detectors. This is due to a lack of reliable benchmarks, and the manual efforts required to validate a large number of candidate clones. In particular, there has been a lack of a synthetic benchmark that can precisely and comprehensively measure clone-detection recall. In this paper, we present a mutation-analysis based benchmarking framework that can be used not only to evaluate the recall of clone detection tools for different types of clones but also for specific kinds of clone edits and without any manual efforts. The framework uses an editing taxonomy of clone synthesis for generating thousands of artificial clones, injects into code bases and automatically evaluates the subject clone detection tools following the mutation analysis approach. Additionally, the framework has features where custom clone pairs could also be used in the framework for evaluating the subject tools. This gives the opportunity of evaluating specialized tools for specialized contexts such as evaluating a tool's capability for the detection of complex Type-4 clones or real world clones without writing complex mutation operators for them. We demonstrate this framework by evaluating the performance of ten modern clone detection tools across two clone granularities (function and block) and three programming languages (Java, C and C#). Furthermore, we provide a variant of the framework that can be used to evaluate specialized tools such as for large gaped clone detection. Our experiments demonstrate confidence in the accuracy of our Mutation and Injection Framework when comparing against the expected results of the corresponding tools, and widely used real-world benchmarks such as Bellon's benchmark and BigCloneBench. We provide features so that most clone detection tools that report clones in the form of clone pairs (either in filename/line numbers or filename/tokens) could be evaluated using the framework.

Index Terms—Clone, Clone Detection, Benchmark, Mutation Analysis, Mutation Operators, Recall

1 Introduction

Code clones are pairs of code fragments within a software system that are similar, either textually, syntactically or semantically. Code reuse by copy and paste, with or without modifications, is one of the most common sources of code clones, although they are known to arise for a variety of reasons [1]. Previous research has shown that clones can be both harmful [1], [2] and beneficial [1], [3], [4] to software quality and the costs of the software development and maintenance. There is a consensus that clones should be detected in order to mitigate their potential harm [2], and to address the other use cases that require the detection of similar code fragments [1], [5].

A 2009 survey of clone detection tools and techniques by Roy et al. [6] found the existence of at least 39 clone detection tools. A 2013 survey by Rattan et al. [7] found at least 70 tools, a 75% increase in only four years. This number reflects the importance the community has placed on clone detection for use in software development and research. However, despite the number of available tools and techniques, there has been a lack of benchmarks for evaluating and comparing their performance.

Clone detection tools are commonly evaluated using recall and precision. Recall is the ratio of the clones within a software system that a tool is able to detect. This measures how well a tool detects and reports true clones. Precision is the ratio of the clones reported by a tool that are true clones, not false positives. This measures the tool's accuracy in detecting and reporting clones. A good clone detection tool has both high recall and precision. While time consuming, clone detection precision can be measured by executing the tool for a variety of software systems from a variety of software domains and validating a significant random sample of the detected clones. This is typically done, at least informally, by the tool's author during development [1]. However, measuring recall is very challenging as it requires foreknowledge of the clones within a subject system(s). As well, decision on whether two code fragments constitutes a true clone is highly subjective [8], and depends on the individual's intended use-case of clone detection [9].

Recall can be measured by building a reference corpus of known clones within a subject system(s), and measuring the ratio of the clones within this reference corpus that the tool is able to detect. This process is extremely effort intensive, as it requires the examination of every possible pair of code fragments in a subject system to determine if they form a true clone pair. Even a small system such as `cook` (51 KLOC, 1244 functions) contains on the order of one million code fragment pairs at the function granularity alone [10].

- *J. Svajlenko is with the Department of Computer Science, University of Saskatchewan, Saskatoon, Canada. E-mail: jeff.svajlenko@usask.ca*
- *C. K. Roy is with the Department of Computer Science, University of Saskatchewan, Saskatoon, Canada. E-mail: chanchal.roy@usask.caof Computer Science, University of Saskatchewan, Saskatoon, Canada.*

This is far too many potential clone pairs to examine accurately, and `cook` does not contain a sufficient number and variety of true clone pairs on its own to properly evaluate a clone detector. Additional subject systems are required, which adds to the workload issue. Instead, clone detection researchers must come up with innovative ways to create benchmarks that can accurately estimate clone detection tool performance without the need to fully oracle multiple subject systems.

Clone detection recall should be evaluated using both synthetic and real-world benchmarking strategies. A synthetic clone benchmark measures the recall of a clone detection tool for artificially constructed clones under controlled conditions. The advantage of a synthetic benchmark is a comprehensive, precise and unbiased measurement of recall at a fine granularity in terms of clone types and cloning contexts. A real-world benchmark consists of real clones mined from a software system or collection of software systems. The advantage of a real-world benchmark is the evaluation of recall for real clones produced by real developers in real software systems. Real-world benchmarks measure recall under real conditions, but precise measurement is more difficult, and a comprehensive and unbiased measurement cannot be guaranteed. Therefore, we get the best evaluation by using both strategies for benchmarking.

The most well-known real-world benchmark is Bellon's Benchmark [11], which Bellon built by manually validating 2% of the clones detected by six contemporary (2002) tools for eight subject systems, requiring 77 hours of manual clone validation efforts. While the union may provide good relative performance evaluation between participating tools [11], there is no guarantee that subject tools have collectively detected all clones within the subject systems and therefore the measure of absolute performance is questionable. The reference corpus is therefore biased by the types of the clones the participating tools detect. We have found that Bellon's Benchmark, and its variants [12], are possibly not suitable for accurately measuring the recall of modern clone detection tools [13]. Baker [14] raised concerns with problems in the creation of Bellon's benchmark, including the clone validation procedure. Charpentier et al. [15] revalidated a number of the clones and found disagreement in the results. While BigCloneBench [16] could overcome some of the concerns of Bellon's benchmark, the reliability of judges in building the benchmark is still a concern. This calls for a synthetic benchmark that could provide a comprehensive and fine-grained analysis of clone detection recall.

In this paper, we present the Mutation and Injection Framework, a synthetic clone benchmarking framework that precisely evaluates clone detection recall at a fine granularity using a mutation-analysis procedure. The framework begins by selecting a random code fragment from a large repository of sample source code. It duplicates and mutates this code fragment to produce a code clone of a known clone type and with a known difference. The mutation operators used in clone synthesis are based on a comprehensive and empirically validated taxonomy of the types of edits developers make on copy and pasted code. The clone is then injected into a software system, evolving the system by a single copy-paste and modify clone. The clone detection tool is then executed for this software system and recall is measured for the injected clone. Since the framework created the clone itself, it is able to precisely evaluate the tool's detection of the clone, including if it appropriately handled the clone-type specific differences between the cloned code. This is repeated thousands of times across all of the edit types in the taxonomy, allowing a comprehensive and exhaustive measurement of recall. The framework fully automates the recall experiment, and allows all aspects of the experiment to be customized and controlled.

The Mutation and Injection Framework has a number of distinct advantages in measuring recall. It supports three programming languages (Java, C and C#) and two clone granularities (function and block) and can evaluate and compare clone detection tools not only for different types of clones but also at a finer granularity. These are abstracted from the procedure, and the framework can be extended to additional languages and granularities with little efforts. It is fully automated, and requires no manual clone validation efforts. The framework includes mutation operators for every type of edit developers make on copy and pasted code. This allows recall to be comprehensively measured at a finer granularity than clone type, allowing a tool's specific capabilities to be measured. The user configures the properties of the clones to be included in the synthesized reference corpus, including clone size, syntactical similarity, mutations and granularity. The user can therefore create a custom benchmark corpus for any general or specific cloning context to evaluate their tool against. Recall experiments produced by the framework can be easily duplicated, shared and modified. Furthermore, we developed a specialized variant of the framework where large-gapped clone detection tools could be evaluated automatically too. In addition to that, the framework has features that let users use custom clones (whether real clones or hand-made) in the injection and evaluation process. This makes it possible to evaluate specialized tools in specialized contexts.

We evaluate the usage and accuracy of the Mutation and Injection Framework by measuring the recall of ten clone detection tools for three languages (Java, C and C#) and two code fragment granularities (block and function) at a fine granularity. This case study demonstrates the compatibility of our benchmark with different clone detectors, and its usefulness in revealing insight about tool performance. In order to build confidence in the accuracy of our benchmark we compare its results against our expectations for the tools, which are based on our knowledge of their capabilities and techniques, and consultations with their original authors where available. In order to gain further confidence on the framework, we compare the results against the existing real-world widely used benchmarks including Bellon's Benchmark [11] and our BigCloneBench [16], [17]. We find strong agreement between the Mutation Framework and both our expectations and the results of BigCloneBench. Among the differences, we do not find any significant anomaly that makes us suspicious of the framework results. For Bellon's benchmark, we experienced disagreements which helped us find anomalies in their benchmark. Our framework results are consistent with a few recent studies [13], [14], [15] that suggests Bellon's benchmark may not be appropriate for evaluating modern clone detectors. We also used the framework for evaluating large gapped clone detection tool.

This paper is organized as follows. We begin with essential definitions in Section 2. We discuss the related work and our previous work in clone benchmarking in Section 3. In Section 4 we fully explore the Mutation and Injection Framework, including how it synthesizes clones, constructs a reference clone corpora, and automates the execution of subject clone detection tools and the measurement of their recall. In Section 5 demonstrate the framework's usage by evaluating ten clone detection tools, and evaluate the effectiveness of our benchmark by comparing its results against our expectations of the tools and their performance measurement by other benchmarks. We discuss the limitations of this work in Section 7 in conclude this work in Section 8.

The Mutation and Injection Framework is open-source and available on-line [18], including the generated benchmarks used in this study.

2 Definitions

Code clones consist of similar source *code fragments*. They are expressed as *clone pairs* and *clone classes*. Unless otherwise stated, we are referring to clone pairs when we use the term "clone". We define these terms as follows:

Code Fragment A continuous region of source code. Specified by the triple (l,s,e), including the source file l, the line it starts on, s, and the line it ends on, e.

Code Clone/Clone Pair A pair of code fragments that are similar, for some type of similarity. Specified by the tuple (f_1, f_2, ϕ), the similar code fragments f_1 and f_2, and their type of similarity, ϕ.

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

Researchers agree upon four primary clone types [1], [11]. The clone types are mutually exclusive, and are defined by the clone detection capabilities needed to identify them. These clone types are defined as follows:

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

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

Type-3 Syntactically similar code fragments that differ at the statement level. The code fragments have statements added, removed, or modified with respect to each other.

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

Clone researchers do not agree upon the minimum syntactical similarity of Type-3 clones, while Type-4 clone detection is not supported by most tools.

3 Related and Previous Work

Some experiments have ignored recall, and simply measured precision by manually validating a small sample of a tool's candidate clones [19], [20], [21], [22], [23]. Others have tackled the recall problem by accepting the union of multiple tools' candidate clones as the reference set, possibly with some manual validation [11], [12], [24], [25], [26]. For some experiments, very small subject systems were manually inspected for clones [27], [28], [29]. An ideal benchmark could be made if all the pairs of code fragments in a subject system were inspected. However, this is not feasible except for toy systems. For example, when considering only clones between functions in the relatively small system Cook, there is nearly a million function pairs to manually inspect [10].

Lavoie and Merlo [30] use of the Levenshtein metric to automatically build Type-3 clone benchmarks. Their methodology avoids the subjectivity of manual clone validation, and builds a complete benchmark that measures both recall and precision. However, the Levensthein metric for clone detection is subject to its own recall and precision. Nonetheless, this methodology is useful for comparing a tool against the Levenshtein metric as a baseline.

Krutz and Le [31] had several expert judges manually validate 1,536 randomly selected method pairs to construct a clone benchmark of 66 true clone pairs across the four clone types. While their benchmark has high confidence, it lacks the size and variety needed to reliable measure clone detection recall.

Yuki et al. [32] built a clone benchmark without manual validation by automatically identifying refactored code clones in the revision history of a software system. The refactoring of these clones by the developers is used as expert validation of the clones. They use code change heuristics and clone similarity metrics to locate instances of clone refactoring by the method merging pattern. Applying this technique across 15,000 revisions of three software projects identified 19 true code clones. A limitation with this technique is it only locates clones that the developers were aware of and were possibly detected using clone detectors, and misses the clones the developers were not aware of.

Schulze and Meyer [33] as well as Ragkhitwetsagul [34], [35] have evaluated clone detection tools for duplicate code that has undergone automatic obfuscation in attempt to hide code plagiarism. This is similar to our mutation framework in that automatic techniques are used to synthesize benchmark clones, although these works have targeted a specific cloning scenario: license violation and plagiarism detection. In contrast, our mutation operators do not target any specific cloning scenario. As our framework is extensible, the obfuscation techniques employed by these researchers could be integrated into the framework as mutation operators, and the Mutation Framework could be used to automatically orchestrate similar experiments.

Wagner et al. [36] built a benchmark of semantically similar program (file) clones by randomly sampling user-submitted solutions to Google Code Jams programming contests. They investigated whether two Type-3 clone detectors and one semantic clone detector could detect these program clones. They reduced their dataset to 58 clone pairs which they publish as an open benchmark. The authors relied on the Google Code Jam process to validate the semantic similarity of the solutions, and did not manually validate them themselves.

Bellon's Benchmark [11] is perhaps the most well-known clone benchmark and there have been several extensions to it as well [12], [13]. It is the product of Bellon et al.'s benchmarking experiment on tools contemporary to 2002 [11].

Their experiment measured the recall and precision of six clone detection tools for eight subject software systems. There are however a number factors that may raise concerns on its accuracy. The union results of the participating tools were used in the construction of the benchmark. While this could give good relative measurements of precision and recall, there is no guarantee that even the whole set of participating tools actually detected all of the clones in the subject systems [14]. Manually building an oracle for a subject system or manually validating a large sample of the detected clones is also challenging [14]. It took 77 hours for Bellon et al. to validate only 2% of the candidate clones in their tool comparison experiment [1]. Manually oracling a small system was possible in some cases [37], [38]. However, even when considering a relatively small system such as `Cook`, and even if we consider only function clones, we would need to examine all pairs of functions of that system, resulting in millions of function pairs to manually check. Even if we had the time available to do that, it would be impossible to do so without some level of inadvertent human error. There is also the question of the reliability and subjectivity of the judges, which neither of these experiments nor the individual tool authors attempted to evaluate. This is a crucial issue in evaluating clone detection tools since even expert judges often disagree in creating clone reference data [39]. Another important aspect is the capability of dealing with different types of clones, in particular, Type 3 or "near-miss" clones, where there could be statement level differences between the cloned fragments, where statements could be added, modified or deleted from the reused pasted fragments. While Bellon et al. had done a great job, an update of the reference corpus was essential [14]. There have been also empirical assessments of the Bellon benchmark. For example, Charpentier et al. [15] also found disagreements in the different types of clones when they were judged by multiple independent judges. In a more recent experiment [8], they also found that the judgments of non-experts could be potentially unreliable. Of course, this is no surprise, since subjective disagreement is an inherent problem in clone detection. While in BigCloneBench [16] we were able to overcome some of these limitations (e.g., no use of any clone detection tools in creating the benchmark and provide type specific clone results), there are still concerns as stated above.

The Mutation and Injection Framework was designed to overcome these limitations. It does away with manual efforts by automatically synthesizing new clones for the reference corpus rather than mining a subject system for existing clones. The artificial clones are based upon a comprehensive taxonomy of the types of edits developers make on copy and pasted code. This ensures that the synthesized clones are both realistic and comprehensive. It avoids subjective clone validation by carefully synthesizing true clones rather than validating potential clones reported by clone detectors themselves. Careful construction of the reference corpus ensures that there are no biases in performance measurement. The Mutation Framework measures recall at a finer granularity than Bellon's Benchmark.

Our previous work includes the proposal of mutation analysis for clone detection tool benchmarking [40], and a proof of concept implementation and experiment to demonstrate its value [41]. The prototype framework was implemented specifically for variants of a single clone detection tool (NiCad [29]), which allowed it to be rapidly implemented.

In this paper we provide a generalized and fully automated version of the Mutation and Injection Framework that can be used to evaluate any clone detection tool. This includes improvements in clone synthesis, matching and corpus validation. We use a rigorous approach, which allows clone detection performance to be measured and compared at a fine granularity without bias. It includes a simple user interface that allows users to customize, share, replicate and extend tool evaluation experiments. A preview of an early version of the full Mutation and Injection Framework was published in a short tool demonstration paper [42]. In this paper, we provide the full details of the framework, as well as an experiment demonstrating its usage and validating its effectiveness.

We have used the framework in previous tool evaluation experiments [13], [17]. We compared the Mutation Framework's results for Java and C block clones against Bellon's Framework [13] to investigate the state of recall measurements of modern clone detectors. We compared the Mutation Framework's results for Java function clones against BigCloneBench [17] to demonstrate the need for both real-world and synthetic benchmarks to fully evaluate clone detection tools. For benchmark comparison we used summary (per-clone-type) results. The tool evaluation experiment in this paper looks at the tool's recall performance at the per mutation-operator (edit-type) granularity for multiple languages and clone granularities.

4 The Mutation and Injection Framework

The Mutation and Injection Framework measures the recall of clone detection tools using a mutation analysis procedure. It is a fully automatic framework that requires no manual efforts during either the construction of the reference corpus nor the evaluation of the subject tools. It achieves this by synthesizing a reference corpus of artificial clones using source-code mutation and injection, rather than mining for real clones in a subject system. An advantage of the framework is that it requires no manual clone validation, which has been an obstacle in measuring recall. By synthesizing the reference corpus, its properties can be controlled, and potential biases can be avoided. An abstract version of the framework is shown in Figure 1, which executes the following procedure:

1) A single clone pair is added to a software system by simulating the creation of a copy, paste and modify clone by a developer. This is accomplished by duplicating and mutating a source code fragment using cloning mutation operators, and introducing this new clone pair into a subject software system using source-code injection.
2) The subject clone detection tool is executed for this mutant version of the subject system.
3) The tool's unit recall is measured specifically for the detection of the injected clone.
4) Steps 1-3 are repeated for a large number and variety of clone pairs.

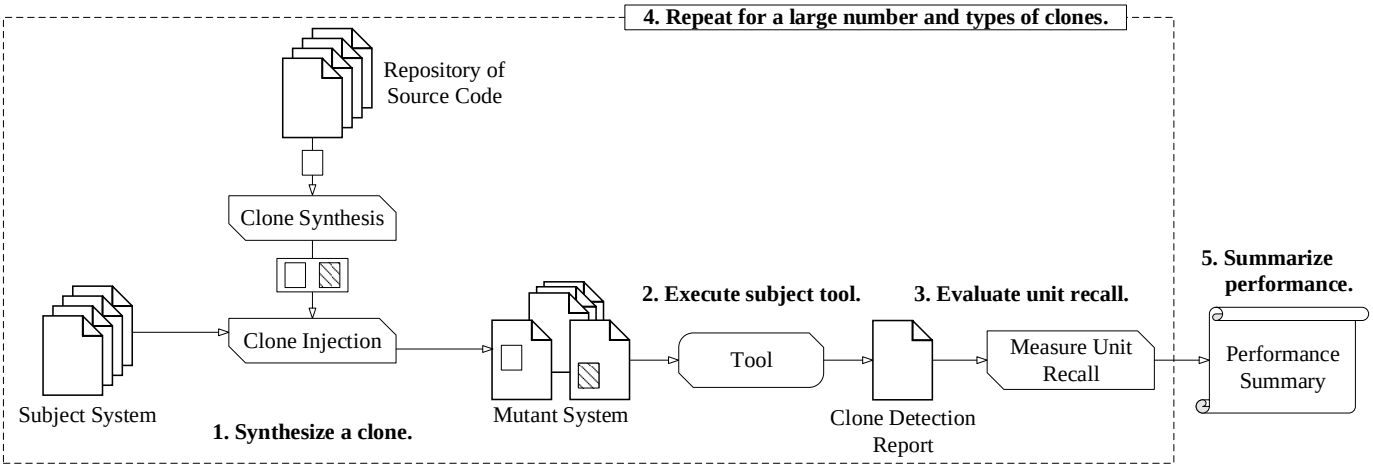

Fig. 1: Overview of the Mutation Framework Procedure

TABLE 1: Editing Taxonomy for Cloning

ID	Edit Description	Clone Type
1	Change in whitespace.	Type-1
2	Change in commenting.	
3	Change in formatting.	
4	Systematic renaming of an identifier.	Type-2
5	Renaming of a single identifier instance.	
6	Change in value of a literal.	
7	Small insertion within a line.	Type-3
8	Small deletion within a line.	
9	Insertion of a line.	
10	Deletion of a line.	
11	Modification of a line.	
12	Reordering of declaration statements.	Type-4
13	Reordering of statements.	
14	Replacement of one type of control statement with another (ex: *for* loop with a *while* loop).	

5) The tool's average recall across these mutant systems (the reference corpus) is reported.

This is a mutation analysis procedure, similar to that used in mutation testing. In mutation testing, mutation operators are used to randomly introduce a bug into a software system and the system's testing strategy is evaluated for its ability to detect and isolate the synthetic bug. To evaluate clone detection tools, we inject duplicate code into a system, and use mutation operators to introduce a random difference between the duplicated code fragments corresponding to one of the clone types. The subject clone detection tool is evaluated for its ability to detect the synthetic clone. The mutation operators are based upon a comprehensive taxonomy of the types of edits developers make on cloned code as shown in Table 1. Details about the taxonomy could be found in our earlier work [6]. This was constructed based upon a literature survey of clone types, clone taxonomies, and empirical studies. This strategy allows us to measure the subject tool's recall not only per clone type but also per clone edit (mutation) type, which provides greater insight into the performance, capabilities and weaknesses of the particular clone detection tool.

Our implementation of the framework splits the abstract procedure into two discrete phases: (1) the generation phase and (2) the evaluation phase. During the generation phase, the clones are synthesized and injected into distinct copies of a subject system. These mutant systems and the injected clones they contain form the reference corpus. The framework allows many constraints to be placed on the generated corpus in order to control its properties. The evaluation phase executes the subject tools for the mutant systems, and measures their recall for the injected clones. The implementation is split in this way to allow the corpus to be generated ahead of time and then shared and reused in multiple tool evaluation experiments.

In the following subsections we describe the framework in full detail. We begin by describing how the clones are synthesized. Next we outline the generation phase, during which clone synthesis is used to create the reference corpus. Then we outline the evaluation phase, and how it automates the subject tool execution and evaluation for the reference corpus.

4.1 Clone Synthesis

The framework synthesizes clone pairs by mutating real code fragments mined from a source code repository. The mutations are based upon the editing taxonomy for cloning (Table 1), which is a validated and comprehensive taxonomy of the types of edits developers make on copy and pasted code. Mutations are applied by *mutation operators*, which take a code fragment as input and output the same code fragment with a single random modification of their edit type. The framework users specify the types of clones to be synthesized using *mutators*, which are sequences of one or more mutation operators. The mutator applies the mutation operators one by one, in order, to an input code fragment. The original code fragment produced by a real developer, and its mutant code fragment produced by a mutator, form a synthetic clone pair produced by mimicking the copy, paste and modify cloning activities of a real software developer. The framework currently supports the synthesis of Java, C and C# clone pairs of the first three clone types at the function and block syntax granularities, although it could be extended to any language.

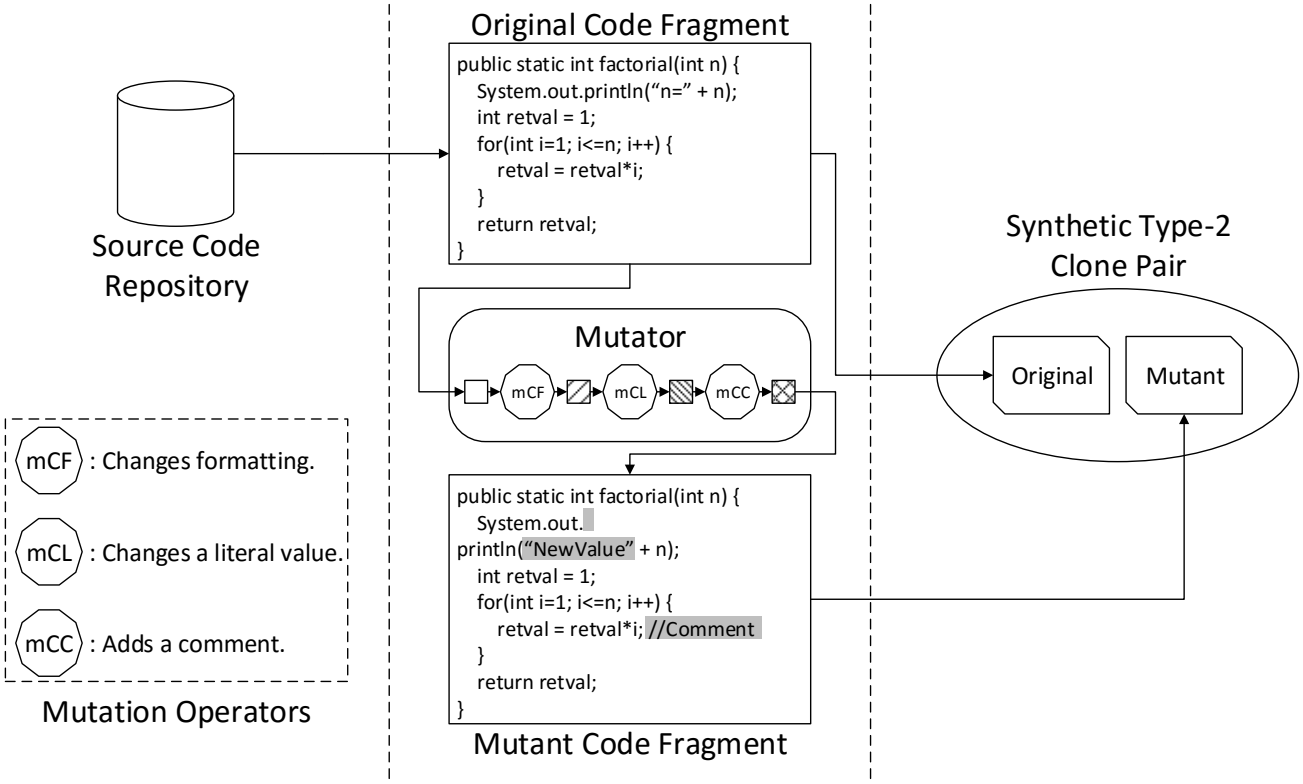

Fig. 2: Clone Synthesis Example

An example of clone synthesis is shown in Figure 2. The original code fragment is mutated by a mutator with a mutation operator sequence of length three. The mutation operators apply a single random change of their edit type. The first mutation operator changes formatting, the second changes the value of a literal, and the third adds a comment. The mutation operators are applied, in this order, to a copy of the original code fragment. The mutations have been highlighted in the final mutant code fragment. The first and third mutation operator apply Type-1 clone differences, while the second mutation operator applies a Type-2 clone difference. Therefore the original and mutant code fragments form a Type-2 clone pair.

In the following sections, we describe the mutation operators and mutators in more detail.

4.1.1 Mutation Operators

The mutations are performed by *mutation operators*, a concept of mutation analysis. From the editing taxonomy, we created fifteen mutation operators that mimic the types of edits developers make on copy and pasted code. These mutation operators take a code fragment as input, and output the same code fragment with a single random edit of the operator's defined edit type. Our mutation operators are summarized in Table 2. This table lists each mutation operators' name, edit type, how the edit type is realized in its implementation, and the clone type the edit belongs to.

The mutation operators are implemented in TXL [43], a source transformation language. They use a simple language-dependent grammar to parse the input code fragment into a syntax tree. The grammar captures both the syntax tokens and the white space (formatting) of the input fragment. The operator's mutation is implemented using a subtree search and replacement pattern. The operator applies the mutation by replacing exactly one randomly selected subtree that matches the search pattern with the corresponding replacement pattern. The output fragment is produced by un-parsing the modified syntax tree. The result is an output code fragment that differs from the input code fragment by a single random application of the intended mutation. The input fragment's syntax and formatting is otherwise unmodified in the output fragment. The operators are able to detect when they can not be applied to a particular input fragment (i.e., when there is no matches to the search and replace pattern), in which case they return an error.

The operators exist as independent programs that are registered with the Mutation Framework. While our mutation operators are registered by default, the user is free to register their own custom mutation operators implemented in any technology they wish. Our mutation operators target general clone detection, and cover our clone taxonomy for the first three clone types. Users of the framework might design their own mutation operators to target specific clone detection use-cases, or for specific targeted debugging of their tool. Custom mutation operators must conform to the same input/output procedure. The mutation operator receives the input code fragment and language as input, and output the mutated code fragment.

The framework includes mutation operators implemented for Java, C and C# source code. Their implementations rely on simple token grammars for these languages.

TABLE 2: Mutation Operators

Name	Edit	Implementation	**Clone Type**
mCW_A	Addition of whitespace.	A tab or space character is inserted between two randomly chosen tokens.	Type-1
mCW_R	Removal of whitespace.	An (syntactically redundant) random tab or space character is removed.	
mCC_BT	Change in between token (/* */) comments.	A /* */ comment is added between two randomly chosen tokens.	
mCC_EOL	Change in end of line (//) comments.	A // comment is added at the end of a randomly chosen line.	
mCF_A	Change in formatting (addition of a newline).	A newline is inserted between two randomly chosen tokens.	
mCF_R	Change in formatting (removal of a newline).	A randomly chosen newline is removed.	
mSRI	Systematic renaming of an identifier.	A randomly chosen identifier, and all of its occurrences, are renamed.	Type-2
mARI	Arbitrary renaming of an identifier.	A randomly chosen single instance of an identifier is renamed.	
mRL_N	Change in value of a numeric literal.	The value of a randomly chosen numerical literal is changed.	
mRL_S	Change in value of a string or character literal.	The value of a randomly chosen string or character literal is changed.	
mSIL	Small insertion within a line.	A parameter is added to a randomly chosen method call or signature.	Type-3
mSDL	Small deletion within a line.	A parameter is deleted from a randomly chosen method call or signature.	
mIL	Insertion of a line.	A line of source code (containing a single statement) is inserted at a random line in the code fragment.	
mDL	Deletion of a line.	A randomly selected line (containing a whole single-line statement) is deleted.	
mML	Modification of a whole line.	A randomly selected line is modified by placing it in a single-line if statement. For example 'x = 15*y;' becomes 'if(X==Y) x=15*y;'	

The mutation operators can be easily adapted to support additional languages. In particular, for C-style languages like Java, C and C#, the mutation operators can be adapted to a new language by simply adding the language's token-based grammar, with no changes the mutation search/replace patterns.

4.1.2 Mutators

The framework user specifies the kinds of clones to be synthesized for the reference corpus using *mutators*, which are sequences of one or more mutation operators. A mutator synthesizes a clone pair by executing its mutation operators as an input/output chain, supplying the input code fragment to the first operator, and retrieving the mutant code fragment from the last mutation operator. Any number of mutators may be defined using the default and custom mutation operators. By specifying the set of mutators used to synthesize clones, the framework user can create a reference corpus tailored for the cloning context they wish to evaluate their subject clone detection tools for.

The mutators also support a number of constraints to be placed on the synthesized clones, including: clone size, clone similarity, and mutation containment. The clone size constraint allows the framework user to specify the minimum and maximum size, by line and by token, of a synthesized clone pair's original and mutant code fragments. The clone similarity constraint allows the framework user to specify the minimum clone similarity, measured by line or by token of the synthesized clones. These constraints can be used to shape the context and properties of the reference corpus, and are optional.

The framework defaults and recommends the use of a single-operator mutator for each of the registered mutation operators. This creates a reference corpus that measures a tool's performance at the edit type granularity. Multi-operator mutators allow a reference corpus with more complex clones to be generated. For example, a mutator set could be defined for a variety of sequences of mutation operators that produce Type-1 clones. This would allow strong evaluation of tools specifically for a variety of Type-1 clones. However, higher order mutations pose some risk. As more operators are applied, it becomes difficult to predict how the operators may interact. Later mutation operators in a mutator's sequence may even reverse previous ones. Additionally, when mutation operators are mixed, you lose the ability to measure performance per edit type. Higher-order mutations are advanced use of the framework, and require careful attention and interpretation. The default single-operator mutators are recommended for standard tool evaluation usage.

4.2 Clone Similarity

Clone similarity (as used in this paper) is the measure of the syntactical similarity between a clone pair's code fragments. It is expressed as a ratio between 0.0 (totally different syntax) and 1.0 (identical syntax). It can be measured by line, by statement, or by language token. The compliment of clone similarity is clone difference, the measure of the syntactical difference between a clone pair's code fragments. For this paper, and with our Mutation and Injection Framework, we measure clone similarity and difference using each code fragment's unique percentage of items (UPI).

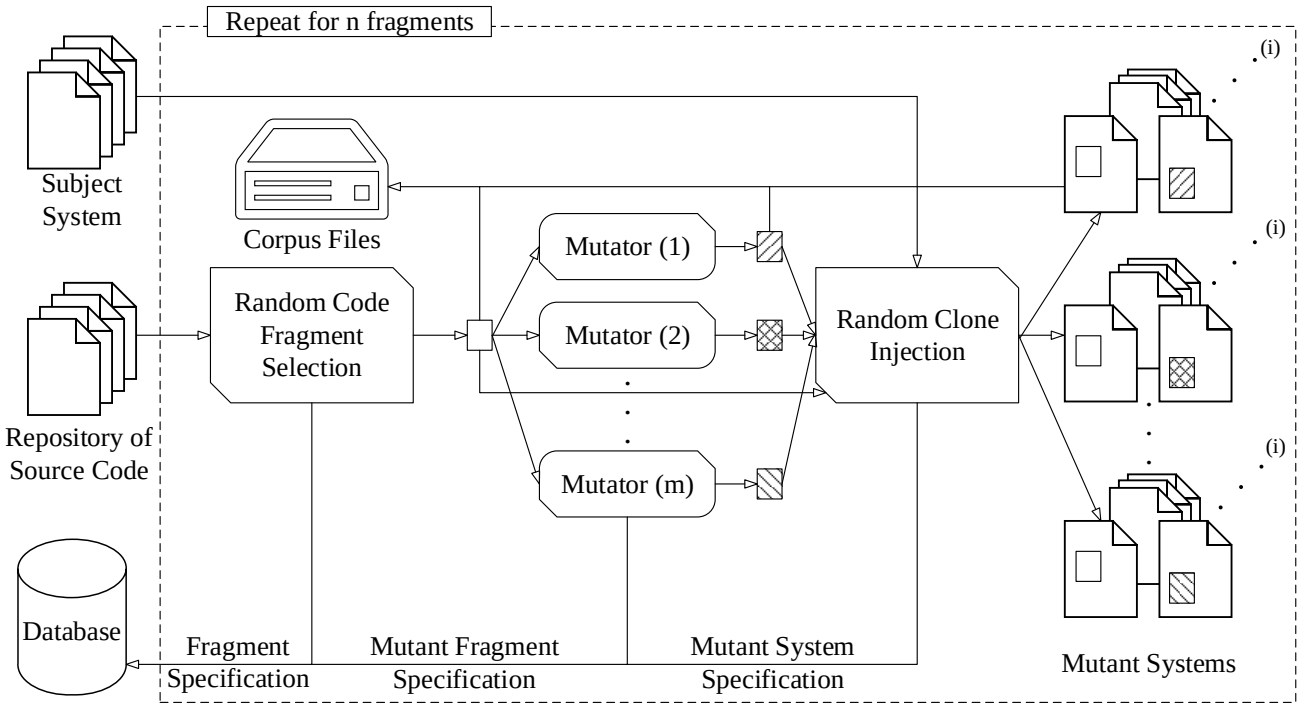

Fig. 3: Overview of Generation Phase

Code fragments can be considered as either sequences of source code lines or language tokens. We can detect the differences in these sequences, from an editing perspective, using the Unix diff program (greatest common subsequences). A code fragment's UPI, with respect to another code fragment, is the ratio of its source code lines or tokens that are not found in the other code fragment, when also considering their order. In other words, the lines/tokens in the code fragment not matched to a line/token in the other code fragment by the greatest common subsequences algorithm. The UPI of code fragment f_1 with respect to code fragment f_2 is expressed mathematically in Equation 1, where *items* can be either source code lines or tokens. We measure clone dissimilarity as the larger of the clone's code fragment UPI, as shown in Equation 2, with clone similarity as its compliment, as shown in Equation 3.

$$upi(f_1, f_2) = \frac{\#\ unique\ items\ in\ f_1\ by\ diff(f_1, f_2)}{\#\ of\ items\ in\ f_1} \quad (1)$$

$$dissimilarity(f_1, f_2) = max(upi(f_1, f_2), upi(f_2, f_1)) \quad (2)$$

$$similarity(f_1, f_2) = 1 - dissimilarity(f_1, f_2) \quad (3)$$

Cloned code fragments often contain Type-1 (white space, formatting and layout) as well as Type-2 (identifier name and literal value) differences. These differences can greatly lower our clone similarity measurement. These types of differences are considered inconsequential in the cloning context, and a more accurate clone similarity is measured if these differences are ignored. We therefore apply Type-1 and Type-2 normalizations to the code fragments before measuring clone similarity. Our Type-1 normalization applies a strict pretty-printing, which results in a single statement per line with normalized whitespace, and removes all comments and blank lines. Our Type-2 normalization replaces each identifier with 'X', and each literal with '0'. With these normalizations, all Type-1 and Type-2 clones will have a similarity of 100%, while Type-3 clones will have a similarity less than 100%. The clone difference will therefore measure the amount of Type-3 differences between the code fragments.

4.3 Generation Phase

During the generation phase, the reference corpus is constructed by synthesizing clones and injecting them into a unique copies of a subject system. An overview of this phase is shown in Figure 3. The generation process begins by selecting a random code fragment from a repository of source code. This selected code fragment is then mutated by m user-defined clone-synthesizing mutators. The resulting m mutant code fragments differ from the selected code fragment by a random application of the mutation defined by their mutator. The mutant code fragments are paired with the selected code fragment to form m synthesized clone pairs. For each of these clone pairs, i mutant versions of the subject system are created by injecting the selected and mutant code fragments into the subject system at a random syntactically correct locations. Each of these mutant systems evolve the subject system by a single copy, paste and modify clone. In total, mi mutant systems are created from a single randomly selected code fragment. This process is repeated for n randomly selected code fragments in order to create a reference corpus containing nmi unique clone pairs. Each reference clone pair is contained within its own mutant

version of the subject system. This is done so the subject tools may be evaluated for each of the reference clones in isolation.

A database is used to track the specification of each selected code fragment, mutant code fragment, and mutant system. The selected and mutant code fragment text is stored in files referenced by the database. This information can be used to construct any of the mutant systems, which contain a single clone from the reference corpus. Clone detection tools can be evaluated for the reference corpus by executing them for each of the mutant systems and evaluating their recall for the injected clone, which is the role of the evaluation phase.

In the following sub-sections we explore the generation phase in detail. We begin by discussing how the generation phase can be configured to control the properties and contents of the generated reference corpus. Next, we detail the steps of the generation phase as they are executed per selected code fragments. There are two primary steps: (1) selecting and mutating a selected code fragment, and (2) injecting the resulting clone pairs into copies of the subject system.

4.3.1 Configuration

The generation phase is highly configurable, which provides the user full control over the size, properties and contents of the generated reference corpus. The user provides a repository of source code from which real code fragments are selected for mutation. The user must also provide the subject software system the synthesized clones are injected into. The user specifies the types of clones to be generated for the corpus by defining the set of clone-synthesizing mutators (Section 4.1) to be used. The generation phase also allows numerous constraints to be placed on the generated corpus, such as the language and granularity of the clones, the size of the corpus, and various clone properties. In this section we describe these configurations and their effect on the generation phase.

Source Code Repository: The user provides a source code repository from which real code fragments are selected for clone synthesis. The repository may be any collection of source code of the target programming language. Ideally, the repository should be large and varied. For example, a combination of the Java standard library and popular 3rd party libraries (e.g., Apache Commons) is a good source code repository for generating a Java clone reference corpus.

Subject Software System: The user provides a subject software system into which the clones are injected. Any subject system of the target language will do, so long as its large enough to have a variety of injection locations. Since each clone in the reference corpus is injected into its own copy of the subject system, the subject clone detection tools have to be executed for mutants of this system many times. Therefore, the subject system should be small enough that the subject tools can be executed for it perhaps thousands of times within a reasonable time frame. We include with the framework special empty subject systems which allow measuring recall for the generated clones in isolation.

Mutator Set: The user can specify any number of mutators by specifying the sequence of mutation operators they should apply to the selected code fragments. The framework's default mutator set includes 15 single-operator mutators, one for each of the 15 default mutation operators. The default mutators mutate the selected code fragment with a single instance of their assigned mutation operator. This creates a reference corpus that can measure the tool's performance at the edit type (mutation operator) granularity.

Generation Constraints: The generation constraints allow the size, scope and clone properties of the generated reference corpus to be specified. The available constraints are as follows:

- **Clone Granularity** The granularity of clones to synthesize. The framework supports clone synthesis at the function and block (a code segment defined by an opening and closing bracket, i.e., {...}) granularities.
- **Language** The programming language of the generated reference corpus. The framework supports the synthesis of Java, C and C# clone pairs.
- **Number of Selected Code Fragments** The maximum number of code fragments to select for clone synthesis.
- **Injection Number** The number of times to inject each synthesized clone pair. In other words, the number of mutant systems to create per clone. Each injection of a clone uses a different injection location in the subject system.
- **Clone Size** The minimum and maximum size of a clone's code fragments. Specified by line and by token, independently.
- **Minimum Clone Similarity** Minimum clone similarity, measured by line and by token using the UPI method described in Section 4.2, after Type-1 and Type-2 normalization.
- **Mutation Containment** The minimum distance the mutation must be from the edges (start/end) of the selected code fragment, specified as a fraction of the fragment's size in lines.

Fragment granularity and *language* are used to set the scope of the experiment. By limiting a corpus to a particular programming language and clone granularity, more specific performance evaluation can be accomplished. The intention is for the user to perform multiple experiments with the framework using the different permutations of clone granularity and language. Performance can then be individually measured per language and granularity.

The *number of selected code fragments*, n, the *injection number*, i, and the size of the mutator set, m, define the maximum size of the generated reference corpus: mni clone pairs. The framework will continue to select code fragments for clone generation until its specified maximum is reached, or all eligible fragments in the repository have been exhausted. A larger number of selected code fragments, n, results in larger diversity in the syntax of the reference corpus's clones. A larger number of injection locations per clone, i, increases the diversity in clone location in the reference corpus's clones. Both are essential for creating a reference corpus that accurately measures recall.

The *clone size* and *minimum clone similarity* constraints make it easier to configure the subject clone detection tools. Most clone detection tools are parameterized by clone size

and some by clone similarity thresholds, which are used to limit the scope of their clone search and reporting processes. These parameters may also affect a tool's execution time and precision. By placing similar constraints on the reference corpus, the tools can be properly and confidently configured for the corpus. This reduces the incidence of inaccurate performance measurement due to configuration mismatch between the tool and reference corpus, which is a significant threat to benchmarking [44]. These constraints may also be used to constrain the corpus for evaluating specific cloning contexts. For example, a corpus could be generated for evaluating the detection of only small or large clones, or a corpus could be made for very similar or less similar clones.

The *mutation containment* constraint is used to ensure that the mutations are an integral part of the synthesized clone pairs. The framework's goal is to measure how well the tools perform for specific differences between cloned code fragments. If the mutation is too near the edge of the clone, it may actually be external to the clone. This constraint ensures that the mutation occurs far enough away from the edges of the clone's code fragments that it is a guaranteed component of the clone pair. And therefore it is correct to require a clone detection tool to handle the mutation to have successfully detected the clone. Mutation containment is specified as a ratio of the size of the selected fragment measured in lines. For a mutation containment of 20% and a selected fragment 10 lines long, all mutant fragments produced will not modify the selected fragment's first or last 2 lines.

The configurable mutators and generation options allow the framework user to generate a variety of well understood benchmark reference corpora. They allow the user to generate a corpus targeting the cloning context they wish to evaluate their tools for. By generating multiple corpora using different configurations, the user can evaluate their tools for a variety of cloning contexts. Having a corpus with exactly known properties ensures that the performance results are correctly interpreted. It also allows the tools to be properly configured for the benchmark.

4.3.2 Step 1 - Code Fragment Selection and Mutation

The first step of the generation phase is to select a real code fragment from a source repository, and mutate it with the m user-defined mutators to produce a set of m synthetic clone pairs. The framework begins by extracting all of the code fragments in the source code repository that satisfy the clone language, granularity and size constraints. The code fragments are tracked by a data structure that allows random selection without repeats.

The data structure is queried for a random code fragment that has not been selected previously. This selected code fragment is mutated by each of the m user-defined mutators, and the output mutant fragments are collected. The mutators are configured to only produce mutant fragments that satisfy the clone size, minimum clone similarity, and mutation containment constraints. A mutator will produce an error if the mutation can not be applied, or if the constraints cannot be satisfied. If all of the mutators are successful, then the selected code fragment is paired with each of its mutant code fragments, and the resulting clone pairs are added to the reference corpus.

If at least one of the mutators fail, for any reason, then the selected fragment and its mutant fragments are discarded, and a new code fragment is selected. This is done to ensure that each mutator contributes the same number of clones to the reference corpus, and to ensure that their clones originate from the same set of selected code fragments. This makes it possible to measure and compare a subject clone detector's recall per mutator without bias due to the code fragment selection or number of synthesized clones per mutator.

This process is repeated until either the maximum number of selected code fragment constraint is reached (not counting the selected code fragments that are discarded), or when all of the code fragments in the repository are exhausted. For a large enough source repository, exhausting the available code fragments should not occur. The generated clones are tracked by a database, including: the origin and text of the selected code fragment, and the mutator and text of its mutant fragments. These clone pairs are now ready for injection into the subject system.

4.3.3 Step 2 - Clone Injection and Mutant Systems

For each of the synthesized clone pairs, one or more mutant systems are created by randomly injecting the clone into the subject system. The mutant system differs from the original subject system by a single copy, paste and modify clone. The injection simulates the development of a new code fragment in the system (injection of the selected code fragment of the clone pair) followed by the cloning and modification of this fragment (injection of the mutant code fragment of the clone pair).

Each clone pair is injected into its own unique copy of the subject system in order to minimize the amount of simulated development performed on the subject system, and to prevent the injected clones from interacting. This allows the framework to evaluate the clone detection tools for each injected clone in isolation, and prevents the properties and structure of the mutant system from diverging too far from that of a real software system.

Clone injection location depends on the clone granularity. Function clones are injected by selecting two random functions in the subject system, and injecting the selected (function) code fragment after one of these functions, and the mutant (function) code fragment after the other. The clone is injected after existing functions rather than before in order to prevent the injection from separating an existing function from its in-code documentation (e.g., javadoc in Java), which is typically placed before the function. Block clones are injected by selecting two random code blocks from the subject system, and injecting the selected (block) code fragment within one of these blocks, and the mutant (block) code fragment within the other. For simplicity, block code fragments are injected either at the start or the end of the chosen code block. A code block can be safely injected at the start or end of an existing code block without creating a syntax error. To simplify tracking of the injected clone, and to prevent any one file in the subject system from diverging too far from its original state, the selected and mutant code fragments are always injected into different source files.

The injection process guarantees that the modified files remain syntactically correct after the injection of the clones.

As the integrity of the generated reference corpus is very important, the framework verifies this by validating the modified source files against a full language grammar. While syntactically correct, the modified files may not compile. The injected code may refer to global variables, fields, functions, types, etc. that do not exist within the subject system. This is not a problem as most clone detectors do not require compiled code, or even compilable code, so long as the code is syntactically correct. Additionally, it is unlikely that the injected code will be semantically compatible with the source files it is injected into. As the framework focuses on syntactical clones and syntactical clone detectors, the semantic mismatch will not affect the tool evaluation.

The *injection number* configuration controls the number of mutant systems the framework creates per generated clone. Each additional mutant system created for a clone will use a different injection location. Using an injection number greater than one is important to the quality of the reference corpus. A clone detection tool may fail to detect an injected clone not due to the syntax of the clone or its clone difference, but due to its location. For example, the tool might be unable to parse the file(s) the clone was injected into due to limitations in its parser, or it may fail to search for clones contained in these files. In this case, the tool may have successfully detected the clone if it had been injected elsewhere. This way, a tool's average recall for different injections of the same clone better reflects it detection of that clone. By varying the injection location, the overall recall of a tool can be more accurately measured.

Injection locations are chosen per selected code fragment rather than per clone. Therefore, all clones originating from the same selected fragment use the same injection location (or set of injection locations if the injection number is greater than one). This way, if the reference corpus is split per mutator, each sub-corpus has the same injection location representation and variance. Recall can then be measured and compared per mutator without bias due to the injection locations used.

In total, mni mutant systems are created, where m is the size of the mutator set, n is the number of selected code fragments, and i is the injection number. The number of mutant systems can be very large. It is not space efficient to store a copy of each of these mutant systems as they differ from the subject system by only the injected clone. The generation phase builds the mutant systems to verify their integrity, but deletes them afterwards. Instead it stores only the specifications of the mutant systems. A mutant system is specified by the clone to be injected (a selected code fragment and one of its mutant code fragments), and the injection location (the source files and file positions to inject at). The mutant systems are then constructed as needed during the evaluation phase using this specification. This keeps the reference corpus small enough for storage on a high performance drive as well as for convenient distribution over the Internet.

4.4 Evaluation Phase

During the evaluation phase, the subject clone detection tools are evaluated for the reference corpus synthesized during the generation phase. The subject tools are executed for each of the mutant systems, and their recall is measured specifically for the injected clones. Remember that the reference corpus contains nmi mutant systems. The n selected code fragments were mutated by m mutators to produce nm clone pairs that were each injected into i copies of the subject system to produce nmi mutant systems, each containing a single injected clone pair. The evaluation phase automates the execution of the subject clone detectors, requiring only a simple communication protocol be implemented for the tool. This protocol, implemented by a *tool runner*, allows the framework to configure and execute the tool, as well as collect and understand its clone detection report.

The evaluation phase is depicted in Figure 4. This processes is repeated per mutant system in the reference corpus. First, the mutant system is constructed from its specification in the reference corpus. Next, the subject clone detection tools are automatically executed for the mutant system using their tool runners. The framework collects and stores the resulting standardized clone detection reports. These reports are analyzed to measure each tool's unit recall specifically for the injected clone in the mutant system.

A unit recall of 1.0 is assigned to a subject tool for a mutant system if the tool successfully detected and reported the injected clone, otherwise it is a assigned a unit recall of 0.0. Successful detection is determined by a clone matching algorithm. This algorithm requires the tool to report a clone pair that: (1) subsumes the injected clone, within a given tolerance, (2) handles the clone-type defining mutation in the injected clone, and (3) exceeds a minimum clone similarity threshold. The reported clone is required to subsume the injected clone, rather than exactly match it, as there may be additional cloned code surrounding the injected clone due to the choice of injection location (at least in the case of block granularity clones). The reported clone must include the mutated portion of the injected clone, as the goal of this framework is to see how well the tools are able to handle these particular differences between cloned code fragments. The reported clone is required to exceed a given similarity threshold to prevent false positives reported by a tool, that happen to subsume the injected clone by chance, from being accepted as a successful match. Therefore, a successful match is a true clone pair that captures the injected clone and handles the particular clone differences introduced by mutation.

The framework measures the subject tools' unit recall for all of the mutant systems. It aggregates these results and reports the subject tools' average recall for various subsets of the reference corpus. Specifically, the framework reports the subject tools' recall per clone type, per mutator, and per mutation operator.

In the following subsections, we describe the evaluation phase in detail. We begin by over-viewing its configurations options. We then describe the subsume-based clone matching algorithm used to determine if an injected clone was detected by a subject tool. Next, we outline the individual steps of the evaluation phase that are executed for each pair of mutant system and subject tool. Including: (1) building the mutant system, (2) executing the clone detection tool for the mutant system, (3) measuring the tool's unit recall for the injected clone. We conclude with an overview of the evaluation phase's final statistical performance reporting

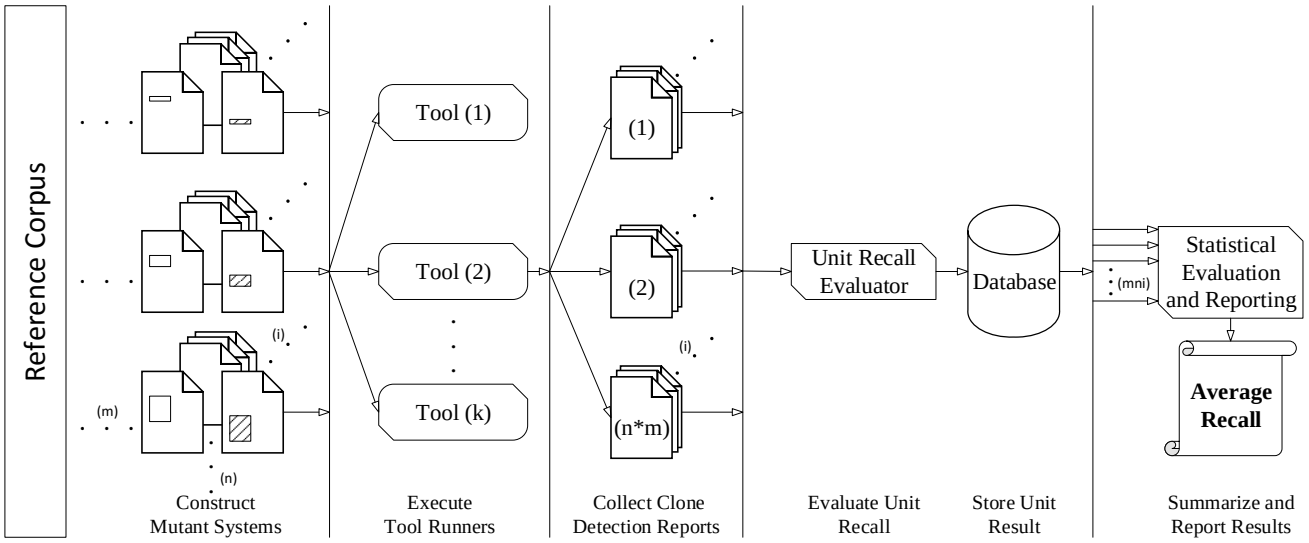

Fig. 4: Overview of Evaluation Phase

about the subject tools.

4.4.1 Configuration

The evaluation phase has the following configuration options listed below. These options are used to configure the subsume-based clone matching algorithm during the evaluation phase. The meaning and consequences of these parameters are explained in the subsequent sections.

Subsume Tolerance A relaxing tolerance for the subsume-based clone matching algorithm, allowing the tool to miss a number of lines from the start and end of a reference clone while still being considered to have subsumed it. Specified as a ratio of the size (measure in lines) of the selected code fragment of the injected clone in the mutant system of interest. Must be equal to or less than the mutation containment in order to require the subject tools to detect the clone-type specific mutant aspects of the reference clones.

Required Clone Similarity The minimum clone similarity of a detected clone to be accepted as a match of a reference clone.

The evaluation phase is also configured with the subject clone detection tools it is evaluating. For each subject tool, the framework requires the following information:

Name The name of the tool.

Description A description of the tool.

Tool Runner The subject tool's tool runner executable.

When a subject tool is registered with the framework, its details are added to the experiment's database, which assigns the subject tool an unique identifier. The name and description of the tool are used by the framework to allow the user to easily identify a subject tool in the performance report. The tool runner is used by the framework to execute the subject tool automatically, and implements the input and output specification expected by the framework. The subject tools are specified by the framework user prior to the execution of the evaluation phase.

4.4.2 Clone Matching Algorithm

The framework uses a subsume-based clone matching algorithm to determine if a given clone pair, C, reported by the subject clone detector for a mutant subject system sufficiently matches the reference clone pair, R, injected into that mutant system, for R to be considered as detected by C. The framework considers C as a sufficient match of R if it meets three criteria: (1) C subsumes R given the configurable subsume tolerance; (2) C captures the mutations in R, and therefore handles the clone-type specific differences in R; and, optionally, (3) C is not likely a false positive that subsumes R by chance, as determined by a clone similarity threshold. We overview these three requirements in detail, followed by a mathematical definition of the full algorithm.

(1) Subsumes the reference clone: To be a successful match, the reported clone (C) must subsume the injected clone (R). In other words, there must exist a pairing of C's and R's code fragments such that the code fragments of C subsume those of R. A subsume-based match is used because the injected clone may be surrounded by additional cloned code due to the selection of injection location. The subject tool may report this larger clone which subsumes the injected clone.

Clone detection tools may not report the reference clones perfectly. In particular, off-by-one line errors are common [11]. For this reason we allow a subsume tolerance to be specified, allowing the tool to miss a number of lines at the start and end of the reference clone while still being considered to have subsumed it. This is controlled by the subsume tolerance configuration, a ratio. The number of lines that can be missed at the start and end of the reference clone's code fragments is equal to the subsume tolerance configuration (a percentage) multiplied by the length (in lines) of the selected code fragment of the injected reference clone, rounded down to the nearest integer. For example, if the injected clone has a selected code fragment 10 lines long, and the framework user specified a subsume tolerance of 15%, then the tool is allowed to miss the first and last 1 lines ($\lfloor 10 * 0.15 \rfloor = \lfloor 1.5 \rfloor = 1$) of the clone. This method

of specifying the subsume tolerance (in lines) is done to consider differences in clone size as well as to accommodate the second criteria.

(2) Handles the clone-type specific differences in the clone: A goal of this framework is to evaluate how well the tools detect clones with particular types of differences, i.e., with different editing activities (mutations) and of different clone types. It is therefore required that the reported clone capture the mutations in the reference clone to be accepted as a successful detection of the reference clone. It is not sufficient for the tool to detect only the identical portions of the clone, it must handle the introduced differences. For example, it is not sufficient for a subject tool to report only the identical regions of a Type-2 clone, it must handle and report the Type-2 differences as part of the overall clone.

This requirement is enforced using the subsume tolerance. During the generation of the reference corpus, the framework user specifies a mutation containment. This is the minimum distance of the mutation from the edges of the selected code fragment, specified as a ratio of the size of the selected (original) code fragment in lines. So long as the subsume tolerance is set equal to or less than the mutation containment, the clone matcher will only accept a reported clone that contains the mutation(s). The framework enforces this relationship between the two configuration parameters to ensure the subject tools must handle the mutations.

(3) Is not a likely false positive (optional): A potential problem with the subsume clone matching algorithm is it will accept any reported clone that trivially subsumes the injected clone, even if the reported clone is a likely false positive. In practice, this is not common (Section 5.2), but could be caused if the tool contains a bug causing it to report line boundaries incorrectly or large selections of mostly dissimilar code as clones. To account for this, we (optionally) require the reported clone to satisfy a minimum clone similarity threshold. This is user configurable, but should be set low as to only filter the obvious false positives. Clone similarity is measured both by line and by token after Type-1 (strict pretty-printing) and Type-2 (identifier and literal) normalizations. This requirement is optional, and should only be used with syntax-based tools where reporting clones with low similarity is unexpected and indicates a bug. Especially for Type-2 clone detectors, where all reported clones should be exact matches after normalization. Of course, similarity thresholds are not perfect at distinguishing false positives, but this limitation can be overcome by running the experiment both with this requirement enabled and disabled as well as using multiple similarity thresholds, which we describe further in Section 4.4.5.

Mathematical Definition: We now summarize the above clone matching algorithm mathematically in Equation 4, where C is a clone reported by the tool, R is the injected reference clone in the mutant system, t is the subsume tolerance, and s is the required clone similarity. The $sim()$ function is implemented as described in Section 4.2. The subsume function is evaluated as in Equation 5, where $C.f_1$ and $C.f_2$ are the code fragments of the reported clone, $R.o$ is the original code fragment and $R.m$ the mutant code fragment of the reference clone, and $T(t) = \lfloor t * R.o.length \rfloor$. Equation 6 defines if a code fragment f_1 subsumes code fragment f_2 given a tolerance of $T(t)$ lines.

$$match(C, R) = subsumes(C, R, t) \wedge sim(C) \geq s \tag{4}$$

$$\begin{aligned} subsumes(C, R, t) = \Big(&C.f_1.subsume(R.o, T(t)) \wedge \\ &C.f_2.subsume(R.m, T(t))\Big) \\ \vee \Big(&C.f_2.subsume(R.o, T(t)) \wedge \\ &C.f_1.subsume(R.m, T(t))\Big) \end{aligned} \tag{5}$$

$$\begin{aligned} f_1.subsume(f_2, T) = \\ &(f_1.file = f_2.file) \wedge \\ &(f_1.startLine \leq f_2.startline + T) \wedge \\ &(f_1.endline \geq f_2.endline - T) \end{aligned} \tag{6}$$

4.4.3 Step 1 - Construct Mutant System

The first step in the evaluation phase process, as executed per subject tool and mutant system pair, is to construct the mutant system. The mutant system's specification is retrieved from the database. This specification references the selected code fragment and its mutant code fragment that comprise the injected clone, along with their respective injection locations. The mutant system is constructed by duplicating the subject system, and copying the code fragments into the specified source files at the specified positions. The mutant system is then ready to be analyzed by the subject clone detection tools.

While each subject tool must be executed for the same mutant systems, the framework deletes and re-constructs the mutant systems for each tool. Many tools leave behind analysis files which could interfere with another tool's execution. A bug in a subject tool could cause changes to the mutant system. To ensure the evaluation is fair, each tool is given a fresh version of each mutant system.

4.4.4 Step 2 - Clone Detection

In this next step, the framework executes the subject tools for the mutant system. The framework is able to execute a subject tool and collect its detection report automatically. To do this, a tool runner must be implemented for the subject tool. A tool runner is an executable that implements a simple communication protocol between the framework and the subject tool. The tool runner wraps the subject tool in an input/output specification that the framework expects. It is the responsibility of the experimenter or tool developer to implement this tool runner.

For a given mutant system, the framework executes the tool runner and passes it the following input parameters: (1) the location of the mutant system, (2) the installation directory of the subject tool, (3) the properties of the reference corpus, including: the minimum and maximum clone size, the minimum clone similarity, and the mutation containment, and (4) the clone type of the injected clone, and the mutation operators used to synthesize it. The tool runner uses this information to configure and execute the subject tool for the mutant system. The framework expects the tool runner to output a clone detection report that lists the clone

pairs the tool found in the mutant system in a simple comma separated format. It is up to the tool runner to covert the subject tool's output format to this standardized format. The framework retains a copy of this clone detection report for evaluation.

The tool runner is free to make use of any of the input data provided to it to configure the subject tool for the mutant system. By implementing multiple tool runners, it is possible to evaluate a tool's performance for different usage scenarios. For example, a tool runner could be implemented that ignores the injected clone information (clone type and mutation operators). It would configure the tool for generic clone detection, and the recall measurements would reflect the general usage of the tool. Another tool runner could consider the injected clone information, and configure itself for targeted detection of the injected clone. This result would be beneficial for tools which are highly configurable, especially towards the detection of specific clone types. The recall measurements would reflect highly targeted clone detection by the tool.

4.4.5 Step 3 - Measuring Unit Recall

Next, the subject tool's detection report is analyzed to evaluate its unit recall for the injected clone in the mutant system. A subject tool is given a unit recall of 1.0 for a particular mutant system if it successfully detects the injected clone, or 0.0 if it does not. To successfully detect the injected clone, the subject tool must report a clone that: (1) subsumes the injected reference clone within a given tolerance (2) captures the clone-type defining mutation in the reference clone, and (3) is itself not an obvious false positive that happens to include the reference clone. These conditions are evaluated by the clone-matching algorithm described in Section 4.4.2. The framework increments through the subject tool's clone detection report until a clone pair is found that is determined to be a successful match of the reference clone by the clone matching algorithm (unit recall = 1.0), or the end of the report is reached (unit recall = 0.0).

The measured recall depends on the configuration of the clone-matching algorithm, including the subsume tolerance and required clone similarity, as discussed earlier (Section 4.4.2). The subsume tolerance defines what ratio of the reference clone the tool can miss from the start and end of the clone while still being considered to have subsumed it. This cannot be set higher than the mutation containment configuration (of the generation phase) to ensure a detected clone is only considered a match of the reference clone if it captures the clone-type specific mutation operators. A higher subsume tolerance is more flexible, allowing the tool to miss more of the clone while still being considered a match, favoring the tools in the evaluation. It acknowledges that when a developer uses the clone detection results, the detected clone is sufficient for them to identify the clone, as well as the missed lines. A lower threshold is more strict with the tools, expecting a more perfect capture. This is important for automated tasks that cannot recognize the missed portions of the clone. The subsume tolerance can also be completely disabled. The framework allows recall to be efficiently measured with multiple subsume tolerances. The framework user can then compare and interpret the differences in measured recall for different strictness in the capture.

The required clone similarity is used to protect against an obvious false positive coincidentally subsuming the reference clone. However, selecting a threshold for indicating obvious false positive clones is too rigid, and will have its own precision problems. To overcome this, the framework supports multiple efficient executions of the evaluation phase using different threshold configurations. For example, recall may be measured for a required clone similarity of: 0%, 50%, 60%, 70%. This is an optional requirement which is appropriate to use with syntax-based clone detectors. We find that disabling this feature still provides high quality results (Section 5.2), and recommend that experiments be run both with this feature enabled and disabled, and compare the results to check for potential impact of limitations of the subsume metric.

Using a threshold of 0% disables this aspect of the clone-matching algorithm. A 50% threshold rejects a reported clone if it shares less than half of its syntax after Type-1 (strict pretty-printing) and Type-2 (identifier and literal) normalizations. A higher threshold is more strict when measuring recall. The measurement can be compared against different thresholds, and the results interpreted. For example, if a tool's recall drops significantly between a 0% threshold (disabled) and 50% (weak threshold), this indicates the tool is capturing (subsuming) the reference clones, but its reporting of the reference clones contains a lot of additional dissimilar code, even after heavy normalization. If recall remains unchanged as the required clone similarity is increased from 0% to the minimum clone similarity of the generated clones (70% for example), this indicates that the tool is both capturing the reference clones and reporting clones that are highly similar syntactically. This threshold cannot be set higher than the minimum clone similarity threshold used in the generation phase. Note that the similarity threshold only affects the validation of Type-3 clones as Type-1 and Type-2 clones have 100% similarity after normalization.

4.4.6 Performance Reporting

The evaluation phase concludes by producing an evaluation report of the subject tools' recall performances for the generated reference corpus. For each tool, the framework reports its recall per mutation operator, per mutator, per clone type, as well as across the entire reference corpus. Summary values are calculated by averaging the unit performances across all mutant systems containing an injected clone part of that summary set. Per mutator performance is calculated by averaging the unit performance across all mutant systems containing an injected clone produced by that mutator. Per mutation operator performance averages the unit performance for all mutant systems containing an injected clone with at least one application of the mutation operator. While per clone type performance averages the unit performance for all mutant systems containing an injected clone of that clone type, as determined by the mutator used.

5 Evaluating the Framework

We have used the Mutation and Injection Framework in a number of studies evaluating and comparing the recall of clone detection tools [13], [17], [45], [46]. In this section, we use results from our previous experiments to demonstrate and evaluate the effectiveness and usefulness of our benchmark. A comprehensive comparison, evaluation and analysis of the available clone detection tools is outside the scope of this paper, and has been a focus of our previous work [13], [17], [45], [46]. Our goal here is demonstration and evaluation of the Mutation Framework itself.

We begin by demonstrating the compatibility of our benchmark with various tools and the usefulness of its results. For this we evaluate the recall of ten clone detection tools at a fine granularity across six benchmarking experiments covering all permutations of the two clone granularities (function and block) and three programming languages (Java, C and C#) supported by the Mutation Framework. We show that measuring recall at the clone edit level (mutation operator) provides new insights into tool performance.

A concern with any benchmark is whether the measured results are accurate, although this is very challenging to evaluate. To build confidence in the results of our benchmark, we compare them against recall measured in three different ways, including: 1) our expectations of the recall of the tools as informed by their features, algorithms and discussions with their authors when available, 2) recall measured by Bellon's benchmark [11], and 3) recall measured by BigCloneBench [16], [17].

5.1 Experimental Setup

Our configuration of the evaluations using the three benchmarks (Mutation Framework, Bellon's Benchmark, BigCloneBench), and the configuration of the tools for these benchmarks, are as follows.

5.1.1 Mutation Framework

Code fragments for clone synthesis were randomly selected from the following source repositories: JDK6 and Apache-Commons (Java), the Linux Kernel (C), and Mono/MonoDevelop (C#). For each experiment, we configured the framework to select 250 random code fragments from the source repository. We used 15 single-operator mutators, one for each of the 15 mutation operators. Each of these mutators apply a single instance of their assigned mutation operator. From the selected fragments, a total of 3,750 unique clone pairs were synthesized by the mutators per language.

We used IPScanner (Java), Monit (C) and MonoOSC (C#) as our subject systems. For each clone, we configured the framework to construct 10 mutant systems using different random injection locations within the subject system. In total, each experiment's reference corpus contains 37,500 mutant subject systems. Across these six experiments, we have constructed 225,000 mutant systems (unique clones) for tool evaluation.

We also inject the clones into empty systems, which consist of two source files with the minimal structural code to allow injection of the clone. For example, for Java this is two source files each with a class with a single empty main function. The empty systems allow us to measure the tools capabilities for the generated clones in isolation as a baseline. Then we can compare how the tools' raw detection of the clones versus when the clone is hidden in a real software system.

We constrained the generation process to give the reference corpora the following clone properties: (1) 15-200 lines in length, (2) 100-2000 tokens in length, (3) a minimum 70% clone similarity measured by line and by token after Type-1 and Type-2 normalization, and (4) a mutation containment of 15%. Since our experiments contain a large number of mutant systems, we preferred slightly larger clones for our reference corpora. Clone detection tools often run significantly faster when configured for larger minimum clone sizes. This allowed us to evaluate the tools in a reasonable time-frame.

We measured unit recall using four minimum clone similarity thresholds: 0%, 50%, 60% and 70%. With 0% the framework measures the tool's ability to capture the injected clone, regardless of the quality of the subsuming reported clone. The non-zero similarity thresholds measure recall with the expectation that the reported clone not only subsumes the injected clone, but contains a minimum degree of syntactical similarity. This is to prevent accepting a match where the candidate clone is suspected to be a buggy clone or a false positive that happens to subsume the reference clone.

We use a range of similarity thresholds as a single threshold may be too rigid. It is difficult to decide which threshold provides the best results. We measure recall for a 0% similarity threshold, to see how well the subject tools capture the reference clones when the quality of the detection is ignored. We use 50% as our weakest definition of a true clone. This requires the code fragments of the true clone pair to share at least half of their syntax, by line or by token. This is a reasonable minimum expectation for a true positive clone of the first three clone types. Our strongest threshold is 70%, which is the minimum similarity of the clones in the generated reference corpora. We also include the 60% threshold as a balance between these two. When we evaluate the tools, we consider how their recall changes as the minimum similarity parameter is varied. This way we overcome the rigidity of a single threshold.

5.1.2 Bellon's Benchmark

We executed the tools for the benchmark's subject systems, and imported their results into the benchmark. We executed the benchmark's mapping and recall evaluation procedures using our *better-ok* clone-matching algorithm, which is an improvement over Bellon's original *ok* metric [13]. This matching algorithm is based on the contains metric, which is shown in Eq. 7.

$$contain(F_A, F_B) = \frac{|F_A \cap F_B|}{|F_A|} \tag{7}$$

The contains metric measures how well code fragment F_B contains code fragment F_A by measuring the ratio of the source lines in F_A that are also in F_B. The *better-ok* clone matching algorithm considers reference clone R

in the benchmark to be matched by candidate clone C reported by the clone detector if the code fragments of C contain the code fragments of R by a given threshold. This algorithm is shown in Eq. 8, where the code fragments are ordered (avoiding comparing both permutations). C is considered as a match for R if the minimum containment of R's code fragments by C's code fragments satisfies the given minimum threshold σ. This algorithm is very similar to the subsume-based clone matching algorithm used with the Mutation Framework, and so it optimal for comparing results between the benchmarks.

$$better\text{-}ok(C, R, \sigma) = min(contain(R.F_1, C.F_1)), \\ contain(R.F_2, C.F_2))) \geq \sigma \quad (8)$$

In contrast, Bellon's original ok clone matching algorithm considers containment in both directions, and will also accept C as a match of R if C is contained by R by the given threshold. The problem with this metric is it accepts poor matches where C is much smaller than R. For example, if R is a 100 line clone, and C is a 5 line clone contained entirely within R, this is considered as a perfect match for R, which is not desirable. Bellon also has a *good* clone matching algorithm based on code fragment overlap, but we found found this metric to be too strict [13]. Murakami et al. [12] extended the benchmarks and matching algorithms to consider the gaps in the Type-3 clones, but we found this had a negligible effect on the recall measured for these tools [13].

We executed the benchmark for a minimum clone matching threshold of 0.70, which is the value used in Bellon et al.'s [11] original experiment, and the value we have used with the Mutation Framework. Further details on the procedure of Bellon's benchmark can be found in its publication [11], and in our previous work [18].

5.1.3 BigCloneBench

BigCloneBench [47], [48] is a collection of 8 million Java function clones of 43 distinct functionalities mined from IJaDataset-2.0, and big Java inter-project repository containing 25,000 open-source software systems. BigCloneBench contains clones of all four clone types, including the entire range of syntactical similarity. BigCloneBench tool evaluation experiments can be conducted using our BigCloneEval tool evaluation framework [48].

Recall was measured using a subsume-based clone-matching algorithm that requires the tool to subsume 70% of a reference clone to be considered to have detected it. This is less strict than the Mutation Framework's clone-matching algorithm. Since the Mutation Framework synthesizes its reference clones, it is able to measure if the tool appropriately handled the clone-type specific mutations it applied, and did not capture the clone by chance within a obvious false positive clone. This is one of the advantages of controlled synthetic benchmarking. While the advantage of real-world benchmarking is measuring recall for natural clones produced by real developers, the clone-matching algorithm cannot be as precise since the clone was not created under controlled circumstances.

We do not include Deckard in this experiment as we previously found its Java-1.4 capable parser is not be sufficient for IJaDataset, leading to poor recall [17]. With BigCloneBench study we use a minimum clone size of 6 lines to match previous real-world benchmark experiments [11], [17], [25]. With the Mutation Framework we used a larger clone size, 15 lines, for performance reasons when executing the subject tools for 225,000 mutant subject systems. However, the results are comparable as we used strict minimum clone sizes and configured the tools appropriately in each case.

5.1.4 Tool Configuration

The participating subject tools, their programming language and clone type support, as well as their their configurations for the three benchmarks are summarized in Table 3.

We configured the tools from an experienced user's perspective. An experienced user has explored a tool's configuration options, defaults and documentation, and modifies the tool's settings, where appropriate, for their use case. An experienced user is not necessarily a clone expert or researcher, but is comfortable configuring the tools for their target input. Specifically, we wanted to measure the performance an experienced user can expect when using these tools with their own subject systems.

To configure the tools we first consulted their documentation and default settings. We enabled any features that provided Type-1 and Type-2 normalization. Clone size and clone similarity thresholds were configured with respect to known properties of the benchmarks. We avoided over-configuring the tools, especially where precision might suffer.

Full Type-1 and Type-2 normalizations were not enabled for some tools. CPD only supports these normalizations for Java subject systems. SimCad has two identifier normalization options: systematic renaming and blind renaming. The blind renaming can find more clones, but can hurt precision, so we used systematic renaming. Simian supports identifier normalization for its supported languages, but we found that it produced unusually large clone reports for C# systems with identifier normalization enables. This indicates a bug or precision problem, so we disabled identifier normalizations with the C# experiments. SourcererCC does not formally support Type-2 clone detection, as it does not use Type-2 normalizations. Instead it targets Type-2 and Type-3 clones using its bag-of-tokens source model and similarity threshold. It does, however, perform stemming on the source tokens which could be seen as a partial Type-2 normalization.

In a couple cases we did not execute a tool for its supported languages or settings. Deckard's Java parser only supports the Java-1.4 specification, so we did not evaluate it in our Java experiments which use the Java-1.6 specification. While Deckard can still be executed for such systems, its detection performance is compromised by the parser. We only executed ConQat for Java. ConQat is a powerful toolkit for rapid development and execution of software quality analysis. Included is functionality for performing clone detection with multiple languages. However, only for Java does it include a preconfigured analysis script for Type-3 clone detection.

TABLE 3: Participating Subject Tools, Their Language and Clone Type Support, and Benchmark Configurations

Tool	Languages	Types	Bellon's Benchmark	Big Clone Bench	Mutation Framework
CCFinderX	Java, C, C#	1,2	min. size: 25 tokens, min. token types: 6	Min length 50 tokens, min token types 12.	Min length 50 tokens, min token types 12.
ConQat	Java, C	1,2,3	Min length 6 lines, max. editing distance 3, gap ratio 30%.	Min length 6 lines, max. editing distance 5, gap ratio 30%.	Min length 15 lines, max errors 3, gap ratio 30%.
CPD	Java, C, C#	1,2	Min length 30 tokens, ignore annotations/identifiers/literals, skip parser errors.	Min length 50 tokens, ignore annotations/identifiers/literals, skip parser errors.	Min length 100 tokens, ignore annotations/identifiers/literals, skip parser errors.
CtCompare	Java, C	1,2	Min length 30 tokens, max 6 isomorphic relations.	Min length 50 tokens, max 6 isomorphic relations.	Min length 100 tokens, max 3 isomorphic relations.
Deckard	Java, C	1,2,3	Min length 30 tokens, 90% similarity, 5 token stride.	Min length 50 tokens, 85% similarity, 2 token stride.	Min length 50 tokens, 85% similarity, 2 token stride.
iClones	Java, C	1,2,3	Min length 30 tokens, min block 10 tokens.	Min length 50 tokens, min block 20 tokens.	Min length 100 tokens, min block 20 tokens.
NiCad	Java, C, C#	1,2,3	Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity.	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	Java, C, C#	1,2,3	Greedy transformation, Unicode support, min 6 lines.	Greedy transformation, Unicode support, min 6 lines.	Greedy transformation, Unicode support, min 15 lines.
Simian	Java, C, C#	1,2	Min length 6 lines, ignore identifiers and literals.	Min length 6 lines, ignore identifiers (Java/C) and literals.	Min length 15 lines, ignore identifiers and literals.
SourcererCC	Java, C, C#	1,3		Min length 6 lines, min similarity 70%, function granularity.	Min length 15 lines, min similarity 70%, function granularity.

5.2 Demonstrating Compatibility and Usefulness

To demonstrate the compatibility of the Mutation Framework, and its usefulness to clone detection researchers and practitioners, we used it to measure the recall of the ten clone detection tools for three programming languages (Java, C and C#) and two clone granularities (block and function). The results of these experiments are summarized in Table 4 and Table 5.

Recall is summarized per mutation operator for two experiments. In our baseline experiment, we measure recall for the generated clones injected into an empty software system (E). This allows us to measure recall for the clones without any bias due to surrounding code which could impact the tools' detection. However, this is not a realistic use-case, so we also measure recall for the generated clones injected into real software systems. For this experiment we configured the clone-matching algorithm to use a 0%, 50%, 60% and 70% required similarity threshold. We show here only the results for 0% (similarity disabled, S0) and 60% (S60) – we found very little difference between 50% and 60%, while 70% was too similar to the similarity threshold used during generation which caused some anomalies. These experiments go from least to most strict detection requirement. We use the $\cdot$ symbol in the results tables to better highlight where recall does not change going from less strict to more strict experiments (left to right). We omit CCFinderX from the baseline experiment as we had difficult getting it to work on a modern Linux system (the other results are from our previous experiment work).

The results of our experiment clearly demonstrate the compatibility of our benchmark. We were able to evaluate ten of the popular and common clone detection tools across three languages and two clone granularities. The gaps in the results are cases where a clone detector does not support a given source language. We did not encounter a case where any of the evaluated tools were incompatible with the framework due to the synthetic nature of the clones or their granularity.

The Mutation Framework is useful in that it measures the recall of clone detection tools for the first three clone types. What sets the Mutation Framework apart from existing benchmarks [11], [16] is it can measure recall per clone edit type (mutation operator) and clone granularity, which leads to unique insights about the tools that is not possible or challenging with existing benchmarks. In the remainder of this section, we will demonstrate this by highlighting unique insights from this experiment. We do not focus on the results of the individual tools, which we have done in previous works [13], [17], [45], [46].

There are a number of reasons why a tool may fail to detect one of the generated clones. (1) It may be unable to detect the clone due to the syntax of the clone originating from the selected code fragment during generation. This will manifest as a uniform drop in recall across the mutation operators as we use the same set of selected code fragments uniformly with each mutation operator. (2) It may be unable to detect a clone due to the clone edit we applied using the mutation operators. This will manifest as a larger drop in recall for the mutation operators the tools' struggle with. (3) A tool may fail to detect a clone not due to properties of the clone, but due to its injection location or the syntax surrounding the clone. This will manifest as a drop in recall going from the base experiment, where the tool is executed against the clone in isolation, to our experiment where the clones are injected into a real software system. We now explore specific instances of these cases and the insights they provide.

We see cases where tools fail to detect clones due to their syntax. CCFinderX, ConQat and Deckard exhibit a uniform recall result across the mutation operators they perform best with.

The Mutation Framework is good at determining which clone types and clone edit types a clone detector supports, which could be different than what the tool advertises or its developers intend (e.g., bugs). The Mutation Framework correctly identifies that CCFinderX, CPD, CtCompare and Simian do not support Type-3 clone detection for any language, and that CPD does not support Type-2 clone

TABLE 4: Mutation Framework Recall Results for Function Clones

			CCFX			ConQat			CPD			CtComp.			Deckard.			iClones			NiCad			SimCad			Simian			Sourc.CC		
			E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60
Java	Type-1	**mCC_BT**	x	99	·	91	·	·	100	99	·	99	97	·				100	·	·	100	·	·	100	·	·	97	91	·	100	·	·
		mCC_EOL	x	99	·	91	·	·	100	99	·	99	97	·				100	·	·	100	·	·	100	·	·	97	91	·	100	·	·
		mCF_A	x	99	·	90	·	·	100	98	·	99	96	·				100	·	·	100	·	·	100	·	·	62	58	·	100	·	·
		mCF_R	x	99	·	91	·	·	0	98	·	99	97	·				100	·	·	100	·	·	100	·	·	70	66	·	100	·	·
		mCW_A	x	99	·	91	·	·	100	99	·	99	95	·				100	·	·	100	·	·	100	·	·	97	91	·	100	·	·
		mCW_R	x	99	·	91	·	·	100	99	·	99	96	·				100	·	·	100	·	·	100	·	·	97	91	·	100	·	·
	Type-2	**mSRI**	x	91	·	88	·	·	100	99	·	99	95	·				85	·	·	100	·	·	100	·	·	97	91	·	100	·	·
		mARI	x	22	·	90	·	·	100	99	·	99	95	·				96	·	·	100	·	·	83	·	·	97	91	·	100	·	·
		mRL_N	x	96	·	91	·	·	0	·	·	0	·	·				99	·	·	100	·	·	100	·	·	96	89	·	100	·	·
		mRL_S	x	99	·	91	·	·	100	99	·	0	·	·				93	·	·	100	·	·	100	·	·	97	91	·	100	·	·
	Type-3	**mDL**	x	0	·	85	·	·	1	·	·	0	·	·				94	·	·	100	·	·	85	·	·	0	·	·	100	·	·
		mIL	x	0	·	85	·	·	0	2	·	0	·	·				98	·	·	100	·	·	90	·	·	0	1	·	100	·	·
		mML	x	0	·	88	·	·	0	·	·	0	·	·				96	·	·	100	·	·	83	·	·	0	·	·	100	·	·
		mSDL	x	0	·	86	·	·	0	·	·	0	·	·				95	·	·	100	·	·	99	·	·	0	·	·	100	·	·
C	Type-1	**mCC_BT**	x	100	·				100	99	98	100	83	·	91	73	·	100	·	·	99	·	·	100	·	·	97	97	·	100	·	·
		mCC_EOL	x	100	·				100	99	98	0	·	·	91	73	·	100	·	·	99	·	·	100	·	·	100	99	·	100	·	·
		mCF_A	x	100	·				100	96	·	100	83	·	91	73	·	100	·	·	99	·	·	100	·	·	63	63	·	100	·	·
		mCF_R	x	100	·				100	96	95	100	83	·	90	72	·	100	·	·	99	·	·	99	·	·	59	59	·	100	·	·
		mCW_A	x	100	·				100	99	98	100	83	·	91	73	·	100	·	·	99	·	·	100	·	·	96	96	·	100	·	·
		mCW_R	x	100	·				100	99	98	100	83	·	91	73	·	100	·	·	99	·	·	100	·	·	97	96	·	100	·	·
	Type-2	**mSRI**	x	98	·				0	·	·	100	80	·	90	72	·	90	·	·	99	·	·	98	·	·	100	99	·	100	·	·
		mARI	x	33	·				0	·	·	100	80	·	91	73	·	99	·	·	99	·	·	92	·	·	100	99	·	100	·	·
		mRL_N	x	100	·				0	·	·	0	·	·	91	73	·	98	·	·	99	·	·	100	·	·	90	90	·	100	·	·
		mRL_S	x	100	·				0	·	·	0	·	·	91	73	·	98	·	·	99	·	·	100	·	·	100	99	·	100	·	·
	Type-3	**mDL**	x	0	·				0	·	·	0	·	·	82	66	·	98	·	·	99	·	·	89	·	·	0	·	·	100	·	·
		mIL	x	0	·				0	·	·	0	·	·	81	65	·	99	·	·	99	·	·	96	·	·	0	·	·	100	·	·
		mML	x	0	·				0	·	·	0	·	·	90	72	·	99	·	·	99	·	·	81	·	·	0	·	·	100	·	·
		mSDL	x	0	·				0	·	·	0	·	·	90	72	·	98	·	·	99	·	·	96	·	·	0	·	·	100	·	·
		mSIL	x	0	·				0	1	·	0	·	·	90	72	·	99	·	·	99	·	·	84	·	·	0	·	·	100	·	·
C#	Type-1	**mCC_BT**	x	100	90				0	51	·										98	·	·	100	·	·	93	88	·	100	·	·
		mCC_EOL	x	100	90				0	52	·										98	·	·	100	·	·	93	88	·	100	·	·
		mCF_A	x	100	90				84	51	·										98	·	·	100	·	·	54	51	·	100	·	·
		mCF_R	x	100	89				0	52	·										98	·	·	100	·	·	53	50	·	100	·	·
		mCW_A	x	100	90				85	51	·										98	·	·	100	·	·	93	88	·	100	·	·
		mCW_R	x	100	90				80	52	·										98	·	·	100	·	·	93	88	·	100	·	·
	Type-2	**mSRI**	x	99	88				0	·	·										98	·	·	100	·	·	0	0	·	100	·	·
		mARI	x	36	33				0	·	·										98	·	·	88	·	·	0	0	·	100	·	·
		mRL_N	x	98	88				0	·	·										98	·	·	100	·	·	91	86	·	100	·	·
		mRL_S	x	100	90				0	·	·										98	·	·	100	·	·	93	88	·	100	·	·
	Type-3	**mDL**	x	0	·				0	1	·										97	·	·	89	·	·	0	1	·	100	·	·
		mIL	x	0	·				0	·	·										98	·	·	93	·	·	0	0	·	100	·	·
		mML	x	0	·				0	·	·										98	·	·	78	·	·	0	0	·	100	·	·
		mSDL	x	0	·				0	·	·										98	·	·	98	·	·	0	0	·	100	·	·
		mSIL	x	0	·				0	·	·										98	·	·	82	·	·	0	1	·	100	·	·

E = Clone only (empty system), S0 = Injected clone, 0% required similarity, S60 = Injected clone, 60% required similarity, · = Same recall as result on left

detection for C. At the edit level, it correctly identifies that CtCompare does not support literal normalization in Type-2 clone detection, and identifies we disabled identifier normalization for Simian with C# due to performance and precision issues.

While CPD advertises literal normalization in Java clone detection, the framework reveals that it does not actually support numerical literal normalization, possibly due to a bug. While CtCompare handles any comment type differences in Java, it unexpectedly is unable to detect clones with differences in end of lines comments in C. Our benchmark is helpful to find such omissions or bugs so that they can be corrected. It would be more challenging to identify these issues using existing real-world benchmarks where the researcher would need to manually examine the clones missed by the tools to try to determine patterns in false negatives. In a real-world benchmark these edit types would be mixed with others which could mislead the researcher, or may lack representation in the benchmark and be missed completely.

The Mutation Framework is good at uncovering particular weaknesses in the tools. For example, while CCFinderX has overall good Type-2 recall, it disproportionately struggles in detecting Type-2 clones with arbitrarily renamed identifies. SimCad is also weakest as these Type-2 clone edits compared to the other edit types, which suggest these isolated changes cause more dissimilarity in its simhashing. In contrast, iClones struggles more with systematically renamed identifiers, as these show up as more dispersed token-level changes to its detection algorithms. While Simian's text-based detection algorithm is robust to most Type-1 clones, it in particular struggles for clones with a change in formatting – where a newline has been added or removed.

Comparing the baseline experiment (empty system) against the primary experiment (real software system), we do see cases where tools are failing to detect clones due to injection location, or other properties of the subject system. We see this in particular for Deckard and CtCompare, and to a lesser degree with CPD and Simian. These tools have

TABLE 5: Mutation Framework Recall Results for Block Clones

			CCFX			ConQat			CPD			CtComp.			Deckard.			iClones			NiCad			SimCad			Simian			Sourc.CC		
			E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60	E	S0	S60
Java	Type-1	**mCC_BT**	x	94	·	94	·	·	100	98	·	99	97	·				100	·	·	100	·	·	100	·	98	96	90	·	100	·	·
		mCC_EOL	x	94	·	94	·	·	100	98	·	99	98	·				100	·	·	100	·	·	100	·	98	96	90	·	100	·	·
		mCF_A	x	94	·	94	·	·	100	98	·	99	98	·				100	·	·	100	·	·	100	·	98	70	66	·	100	·	·
		mCF_R	x	94	·	94	·	·	0	98	·	99	98	·				100	·	·	100	·	·	100	·	98	60	55	·	100	·	·
		mCW_A	x	94	·	94	·	·	100	98	·	99	98	·				100	·	·	100	·	·	100	·	98	96	90	·	100	·	·
		mCW_R	x	94	·	94	·	·	100	98	·	99	98	·				100	·	·	100	·	·	100	·	98	96	90	·	100	·	·
	Type-2	**mSRI**	x	90	·	92	·	·	100	98	·	99	96	·				84	82	·	100	·	·	100	·	98	96	90	·	100	·	·
		mARI	x	20	·	94	·	·	100	98	·	99	96	·				96	98	·	100	·	·	91	·	89	96	90	·	100	·	·
		mRL_N	x	93	·	94	·	·	0	·	·	0	·	·				98	98	·	100	·	·	100	·	97	93	88	·	100	·	·
		mRL_S	x	94	·	94	·	·	100	98	·	0	·	·				94	97	·	100	·	·	100	·	97	96	90	·	100	·	·
	Type-3	**mDL**	x	0	·	87	·	·	1	·	·	1	·	·				94	·	·	100	·	·	94	·	91	2	2	·	100	·	·
		mIL	x	0	·	87	·	·	2	·	·	0	·	·				96	97	·	100	·	·	96	·	93	1	1	·	100	·	·
		mML	x	0	·	90	·	·	0	·	·	0	·	·				95	97	·	100	·	·	88	·	86	0	0	·	100	·	·
		mSDL	x	2	·	89	·	·	0	·	·	0	·	·				95	95	·	100	·	·	99	·	96	0	0	·	100	·	·
		mSIL	x	2	·	90	·	·	1	·	·	1	·	·				97	98	·	100	·	·	89	·	87	0	0	·	100	·	·
C	Type-1	**mCC_BT**	x	99	·				100	·	·	100	81	·	90	61	59	100	·	·	100	·	·	100	·	98	96	93	·	100	·	·
		mCC_EOL	x	99	·				100	·	·	0	·	·	90	61	59	100	·	·	100	·	·	100	·	98	100	97	·	100	·	·
		mCF_A	x	99	·				100	·	·	100	81	·	90	61	59	100	·	·	100	·	·	100	·	98	64	61	·	100	·	·
		mCF_R	x	99	·				100	·	·	100	81	·	89	61	58	100	·	·	99	·	·	99	·	97	54	53	·	100	·	·
		mCW_A	x	99	·				100	·	·	100	81	·	90	61	59	100	·	·	100	·	·	100	·	98	97	94	·	100	·	·
		mCW_R	x	99	·				100	·	·	100	81	·	90	61	59	100	·	·	100	·	·	100	·	98	98	95	·	100	·	·
	Type-2	**mSRI**	x	98	·				0	·	·	100	81	·	90	61	59	96	95	·	100	·	·	100	·	97	100	97	·	100	·	·
		mARI	x	30	·				0	·	·	100	81	·	90	61	59	100	99	·	100	·	·	96	·	93	100	97	·	100	·	·
		mRL_N	x	99	·				0	·	·	0	·	·	90	61	59	100	99	·	100	·	·	100	·	98	88	85	·	100	·	·
		mRL_S	x	99	·				0	·	·	0	·	·	90	61	59	97	96	·	100	·	·	100	·	98	100	97	·	100	·	·
	Type-3	**mDL**	x	0	·				0	·	·	0	·	·	78	57	54	98	97	·	99	·	·	91	·	88	0	0	·	100	·	·
		mIL	x	0	·				1	·	·	1	·	·	81	58	56	98	98	·	100	·	·	94	·	92	1	1	·	100	·	·
		mML	x	1	·				1	·	·	1	·	·	89	61	58	98	97	·	100	·	·	88	·	86	0	0	·	100	·	·
		mSDL	x	0	·				0	·	·	0	·	·	89	60	58	98	98	·	100	·	·	98	·	95	0	0	·	100	·	·
		mSIL	x	1	·				1	·	·	1	·	·	90	61	59	98	98	·	100	·	·	91	·	89	0	0	·	100	·	·
C#	Type-1	**mCC_BT**	x	96	96				0	50	·										99	·	·	97	·	·	86	84	·	100	·	·
		mCC_EOL	x	96	96				0	51	·										99	·	·	97	·	·	87	84	·	100	·	·
		mCF_A	x	96	96				92	51	·										99	·	·	97	·	·	56	53	·	100	·	·
		mCF_R	x	96	96				0	51	·										98	·	·	96	·	·	50	49	·	100	·	·
		mCW_A	x	96	96				86	50	·										99	·	·	97	·	·	88	84	·	98	·	·
		mCW_R	x	96	96				78	51	·										99	·	·	97	·	·	88	84	·	100	·	·
	Type-2	**mSRI**	x	94	94				0	·	·										99	·	·	97	·	·	0	·	·	100	·	·
		mARI	x	35	35				0	·	·										99	·	·	85	·	·	0	·	·	100	·	·
		mRL_N	x	94	94				1	0	·										99	·	·	97	·	·	86	83	·	100	·	·
		mRL_S	x	96	96				0	·	·										99	·	·	97	·	·	86	82	·	100	·	·
	Type-3	**mDL**	x	2	2				1	0	·										98	·	·	83	·	·	1	·	·	100	·	·
		mIL	x	0	0				0	·	·										99	·	·	90	·	·	0	·	·	99	·	·
		mML	x	0	0				0	·	·										99	·	·	83	·	·	0	·	·	100	·	·
		mSDL	x	0	0				0	·	·										99	·	·	94	·	·	1	·	·	100	·	·
		mSIL	x	0	0				0	·	·										99	·	·	82	·	·	0	·	·	100	·	·

E = Clone only (empty system), S0 = Injected clone, 0% required similarity, S60 = Injected clone, 60% required similarity, · = Same recall as result on left

good detection of the clones, but lose recall when these clones are injected into various positions in a real software system. While clone injection is causing a bias in this case, injection represents a more realistic use-case of these tools. The baseline represents the performance of these tools in the ideal case where the software system has no impact on their performance, but the primary experiment may reflect better the actual performance of these tools in real use-cases. We see an anomaly with CPD for C# clones where it has better recall in the baseline case for some kinds of Type-1 clones, but for others it is only able to detect them when injected into a real software system, suggesting a bug with the tool. ConQat, iClones, NiCad, SimCad and SourcererCC are not impacted by injection location as revealed by no change between the baseline and primary (at least S0) experiments.

The Mutation Framework's flexible similarity requirement for the clone matching algorithm allows clone detection quality to be considered. The majority of the tools show little or not difference in recall when the similarity requirements is increased from 0% to 60%. Only CCFinderX and only for function granularity C# clones do we see a significant difference, with an absolute drop of up to 11%. Since CCFinderX is a Type-1/Type-2 clone detector, and should not be reporting clones with a similarity less than 100%, this suggests a issue in reporting clone boundaries for the C# function clones. While we cannot rely on the required similarity threshold to judge true vs false clones, we do find it is useful to find cases where limitations in the subsume-matching metric may be impacted by bugs in the clone detector.

The Mutation Framework is also good at determining if a clone detection tool has a weakness for a particular language or granularity. While most of the tools have similar recall for block and function clones, we note that Deckard has significantly higher recall for function clones than block clones. CCFinderX has somewhat lower recall for function clones in C#, and somewhat lower recall for block clones in Java, but the difference is not as pronounced as for Deckard.

Differences in function and blocks recall could indicate some issue in how these tools align the code fragments for clone detection. Across the programming languages, we see that CtCompare performs better for Java than C, and CPD performs vastly better for Java and C than for C#. Simian has marginally higher recall for Type-1 C clones compared to Type-1 Java clones. Where recall is lower for one language could indicate problems in that parser for that language.

In this experiment, we have demonstrated that the Mutation Framework can provide insights about the tools that are only possible with its fine-grained analysis. While real-world benchmarks contain clones with the various clone edit types and granularities, they are not able to evaluate the tools for them in isolation with limited bias. The Mutation Framework is symmetric (selected code fragments, injection locations) across the mutation operators, allowing fine-grained recall to be compared across the edit types with reduced bias.

5.3 Building Confidence in Benchmarking Accuracy

While the Mutation Framework was created to measure the recall of clone detection tools, it is very challenging to measure the accuracy of the Mutation Framework itself. In order to build confidence in recall results measured by our Mutation Framework we evaluate the tools in three additional ways and compare these results against our benchmark to build our confidence and to check for any anomalies in the results. First we compare the results against our expectations for the tools based on their algorithms, features and consultations with their authors. While we expect to find disagreements with our expectations where the Mutation Framework provides new insights into the tools, we do not expect to find significant anomalies between our knowledge of the tools and the benchmark results. We then compare our synthetic benchmarking results against measurements from two real-world benchmarks, Bellon's Benchmark and BigCloneBench. Strong agreement between benchmarks built in different ways builds our confidence in their accuracy. While we expect to find differences due to the advantages and disadvantages of the different benchmarking methodologies, these differences should be explainable by the differing approaches. Anomalous disagreement between the benchmarks suggests inaccuracies in one or both of the benchmarks. We use our Mutation Framework results for injection into a real software system and with a 60% similarity threshold as the strictest experiment and the most similar to the other benchmarks.

5.3.1 Mutation Framework vs. Expectations

We build confidence in the Mutation and Injection Framework by comparing its results against our knowledge and expectations of the evaluated tools. While we cannot be certain our expectations are correct, and agreement between our expectations and our benchmark does not guaranteed the Mutation Framework is accurate, this comparison is a good "sanity check" for problems in the benchmark. Significant or anomalous disagreement between our knowledge/expectations and the benchmark results could indicate problems in the benchmark, therefore we investigate the cases of disagreement carefully to check for problems.

We show the agreement between our expected recall and the measured recall by the Mutation Framework in Table 6. We compare the Mutation Framework's agreement with our expectations separately for the function and block clone granularities, although we did not create expectations separately for these granularities. Agreement is marked with a crossed circle, while disagreement is marked with a empty circle. The table summarizes the ratio of the cases where the framework and our expectations agree.

Our expectations of the subject tool's recall are shown per language and clone type in Table 6. Our expectations were decided before the experiments were executed. We evaluated our expectations of the tools' recall in 25% increments, starting at 0% and capped at 90%. We consider a measured recall to agree with our expectation if it is within 12.5% of the expected value. This strategy gives our expectations flexibility, as they are educated estimates.

If agreement is found, then we are confident that the expectation and benchmark are correct. Otherwise, we suspect that either our expectation and/or the benchmark is inaccurate. We chose our expectations by consulting the tools' documentation, publication, and literature discussion [6], [7]. We also considered our experiences with these tools in our other studies. Where possible, we reached out to the tool developers for their opinions on our expectations. In general, we were optimistic about the quality of the tools. Despite these efforts, the expectations may still contain inaccuracies. This is why we use a generous window (25%) around the expectation when determining agreement.

We expect a recall of 0% if a tool does not support the detection of a particular clone type. If a tool fully supports a clone type (e.g., supports the required source normalizations), we assigned it an expected recall of 90% for that type. If a tool only partially supports a clone type, or if we felt it may struggle in detecting a clone type, we estimated the degree of its partial support as 25%, 50% or 75%.

The Type-3 clone detectors we tested use advanced detection techniques and support a variety of source normalizations. We expect that these tools have high recall (90%) for the three clone types. We expected the Type-2 clone detectors (CCFinderX, CPD, CtCompare, Simian) to have 0% Type-3 recall as they do not support clones containing gaps.

CPD only supports Type-2 normalizations with Java source code, so we expected it to have 0% recall for Type-2 clones in C and C# systems. With CtCompare, we lowered our Type-2 expectations to 50% as CtCompare does not support literal normalization. We lowered SourcererCC's expected Type-2 recall to 75% because SourcererCC does not apply Type-2 normalizations other than token stemming. However, its Type-3 detection capabilities, a flexible bag-of-tokens model and similarity threshold, should allow it to detect many Type-2 clones when considering the Type-2 differences as Type-3 gaps. We lowered our expectations of Deckard's Type-3 recall to 75%. Deckard recommends a relatively higher similarity threshold to maintain precision. This may cause it to miss some Type-3 clones with larger degrees of dissimilarity.

The expected and measured recall are compared in Table 6. Since we selected our expectations in 25% increments,

TABLE 6: Agreement Between Measured and Expected Recall

	Tool	CCFX			ConQat			CPD			CtCom.			Deckard			iClones			NiCad			SimCad			Simian			Sourc.CC			%Agree
	Clone Type	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	
Java	Expected	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◑	○	—			◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	◕	◕	◕	-
	Function-Actual	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◑	○	—			◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	◕	◕	◕	-
	Function-Agree	⊗	○	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	⊗	—			⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	89%
	Block-Actual	◕	◕	○	◕	◕	◕	◕	◕	○	◕	◑	○	—			◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	◕	◕	◕	-
	Block-Agree	⊗	○	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	⊗	—			⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	89%
C	Expected	◕	◕	○	—			◕	○	○	◕	◑	○	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	◕	◕	◕	-
	Function-Actual	◕	◕	○	—			◕	○	○	◕	◑	○	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	◕	◕	◕	-
	Function-Agree	⊗	⊗	⊗	—			⊗	⊗	⊗	○	⊗	⊗	○	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	85%
	Block-Actual	◕	◕	○	—			◕	○	○	◕	◑	○	◑	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	◕	○	◕	◕	◕	-
	Block-Agree	⊗	⊗	⊗	—			⊗	⊗	⊗	○	⊗	⊗	○	○	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	81%
C#	Expected	◕	◕	○	—			◕	○	○	—			—			—			◕	◕	◕	◕	◕	◕	◕	◑	○	◕	◕	◕	-
	Function-Actual	◕	◕	○	—			◑	○	○	—			—			—			◕	◕	◕	◕	◕	◕	◕	◑	○	◕	◕	◕	-
	Function-Agree	⊗	⊗	⊗	—			○	⊗	⊗	—			—			—			⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	○	⊗	83%
	Block-Actual	◕	◕	○	—			◑	○	○	—			—			—			◕	◕	◕	◕	◕	◕	◕	◑	○	◕	◕	◕	-
	Block-Agree	⊗	⊗	⊗	—			○	⊗	⊗	—			—			—			⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	○	⊗	83%

⊗ = Agree (±12.5%) ○ = Disagree ○ ◔ ◑ ◕ ● = Recall (0, 25, 50, 75, 90%)

we consider a measurement to agree with our expectations if they are within ±12.5%. The framework agrees with our expectations in 81-89% of the cases, depending on the programming language and clone granularity. This high agreement allows us to have confidence in the results of our benchmark. However, there is some disagreement. This could be due to errors in our expectations, or inaccuracies in the benchmark. We investigate these cases to determine the likely reason. With the exception of SourcererCC's Type-2 recall, measured recall was lower than our expectations in the cases of disagreement.

In most of the cases, disagreement was due to tool having an unexpectedly low recall for one or two particular mutation operators (i.e., types of differences between cloned code). This would cause the tools' average recall for the corresponding clone type(s) to fall below expectations. The tools' recall for the other mutation operators of that clone type agreed with the expectations. This scenario covers the cases of disagreement with CCFinderX, CPD (for Java), CtCompare and Simian. Each of these tools have difficulties with different mutation operators, so there does not appear to be any systemic issue with how the framework handles these mutations. At least one tool performs well for the clones produced by each mutation operator. Rather, disagreement appears to be due to an incorrect expectation. These tools contain deficiencies that were unknown to us. We are therefore confident that for these cases of disagreement, the Mutation Framework is correct and our expectations are incorrect.

With SourcererCC, the Mutation Framework has measured a higher Type-2 recall than expected. We believed SourcererCC may miss some Type-2 clones because it does not use full Type-2 normalization, only language token stemming. SourcererCC does not use Type-2 normalization as it would cause poor precision and execution performance with its optimized bag-of-tokens approach. However, it appears that the dissimilarity caused by the Type-2 changes fall within the Type-3 clone similarity threshold of the tool, allowing the tool to detect them using its Type-3 detection capabilities.

Disagreement with expectations for Deckard and CPD (for C#) do not appear to be due to a inefficiency with any particular kind of clone. Rather these tools' recall for every mutation operator was lower than expected. It is possible our expectations were too high for these cases of disagreement, or it may be that the Mutation Framework is measuring a recall that is too low. However, since the other cases support confidence in the Mutation Framework, and we do not see uniformly poor recall across all the tools (some tools are detecting these clones well), we therefore believe our expectations were too high in these particular cases.

The Mutation Framework agrees with our expectations in a large majority of the cases. In most of the cases of disagreement, we have found specific unknown weaknesses in the subject tools. Overall, we did not find any major or consistent anomalies in the results of the Mutation Framework that would lead is to suspect its accuracy. While this comparison does not prove the accuracy of the Mutation Framework, it does build our confidence that the benchmark is providing accurate measurements.

5.3.2 *Mutation Framework vs. Bellon's Benchmark*

Now we compare the Mutation and Injection Framework (synthetic benchmark) against Bellon's Benchmark (real-world benchmark). Since Bellon's benchmark contains clones at the arbitrary line level, we compare against the block granularity results from Mutation Framework as the closest match. Recall per clone type and programming language as well as agreement between the benchmarks is shown in Table 7. We consider the benchmarks to agree if their measurements have an absolute difference no larger than 15%, which is a value we have used successfully in our previous work [13]. As can be seen, there is poor agreement between the benchmarks. The benchmarks agree for only 38% of measurements for Java, and 25% for for C. The benchmarks agree that the tools that do not support Type-3 detection have poor or no detection of these clones. Otherwise the benchmarks have some agreement on CPD, CtCompare, iClones and Simian. Where the benchmarks

disagree, Bellon's benchmark consistently measures a lower recall, often significantly lower.

Some disagreement is expected between the benchmarks. The Mutation Framework generates only simple Type-3 clones with a single change, so Bellon's benchmark may measure a lower Type-3 recall where clones can have more significant differences. We expected Bellon's benchmark to measure lower recall for NiCad and SimCad due to clone granularity. These tools only detect clones at the strict function and block granularity, whereas Bellon's benchmark contains clones at arbitrary granularity. This is a tool compatibility issue with Bellon's benchmark, which is a problem since many recent clone detectors have targeted strict (usually function) granularities [45], [46], [49], [50], [51]. In contrast, clone detectors that report clones at arbitrary granularity are compatible with the function and block clones produced by our Mutation Framework.

Where disagreement is most puzzling is for Type-1 and Type-2 recall. The Mutation Framework shows us that many of these tools have high recall for the various kinds of Type-1 and Type-2 clone edits. We expect this would translate to high recall for real clones in real software systems, and yet Bellon's Benchmark is measuring low Type-1 and Type-2 recall for many of these tools. These tools feature the normalizations required to detect the Type-1 and Type-2 clones, and the Mutation Framework confirms they work correctly, so this leads us to be suspicious of the results from Bellon's Benchmark.

Analysis of Bellon's Benchmark from the literature supports our suspicions of Bellon's Benchmark. Baker [14] found inconsistencies in Bellon's validation of the clones. Charpentier et. al [15] re-investigated some of Bellon's clones with additional judges and found disagreement in the validation results. Our previous extensive evaluation of Bellon's Benchmark [13] found that the benchmark may not be applicable to modern clone detectors. For example, we compared the evaluation of the modern CCFinderX with the benchmark against the results for the contemporary CCFinder (which participated in Bellon's original experiment) and found the benchmark measured significantly poorer recall for CCFinderX. A significant obstacle with Bellon's benchmark is it was built using the clone detectors themselves, and it may not be accurate for measuring the recall of clone detectors that did not contribute to the benchmark. Updating the benchmark would require significant effort without solving the root issues, and expanding the benchmark is not possible as Bellon's clone validation procedure is not sufficiently documented [14], [15].

Strong disagreement between these benchmarks suggests that one or both are incorrect. The anomalies in the Bellon's benchmark results for the modern tools makes us suspicious of its results, but does not guarentee the Mutation Framework is accurate. An extensive evaluation of Bellon's benchmark can be found in our previous work [13], which explores additional anomalies in the Bellon's benchmark results for modern tools. Our conclusion was that an updated real-world benchmark was needed to evaluate the modern tools and to support the accuracy of our Mutation Framework. This was one of our primary motivations to build BigCloneBench [16].

5.3.3 Mutation Framework vs. BigCloneBench

We introduced BigCloneBench [16] as a real-world benchmark for evaluating modern clone detection tools in order to at least partially overcome the limitations of Bellon's benchmark. Now we compare its results against our Mutation Framework to build confidence in the accuracy both benchmarks. We expect that when the Mutation Framework measures good performance for the various types of clone edits that this should translate into good real-world performance as measured by BigCloneBench. Since these benchmarks were built using very different methodologies, agreement builds our confidence in the accuracy of both benchmarks, whereas disagreement could suggest an anomaly in one or both benchmarks. We measure agreement between the benchmark for the first three clone type, and investigate reasons for disagreement.

We do expect some disagreement between the benchmarks due to the nature of synthetic vs real-world benchmarks. We expect recall to be lower for BigCloneBench, where the real-world clones are more complex and contain various combinations of clone edits. Differences in recall measured per clone type may be different in BigCloneBench where the distribution of clone edits reflect real systems, whereas the Mutation Framework uses a flat distribution to summarize per-type recall from the per-mutation results. Since the Mutation Framework generates simple clones with known properties, its clone matching algorithm is more precise and can check that the tools correctly detected the type-specific clone edit within a reference clone. BigCloneBench cannot be that precise since the clones are more complex, and might measure higher recall for the clone types that the clone detectors do not formally support.

Since BigCloneBench contains Java function clones, we compare it against the Java Function recall measured by the Mutation Framework. BigCloneBench evaluates recall for clones across the entire range of syntactical similarity, and reports Type-3 recall for different regions of syntactical similarity. We compare the Mutation Framework's Type-3 recall measurement against BigCloneBench's 'Very-Strongly Type-3' recall measurement. This is Type-3 recall for clones with a syntactical similarity, after both Type-1 and Type-2 normalizations and measured both by line and by token as shown in Section 4.2, in the range: $[90, 100)\%$. We find that Type-3 clones in this range best match the Java function clones in our Mutation Framework experiment. While the Mutation Framework was configured to generate clones with a minimum 70% Type-3 clone similarity, the single-operator Type-3 mutators mostly produced very similar Type-3 clones. Type-3 recall at lower syntactical similarities can be found in our previous BigCloneBench experiment [17].

Recall is summarized per clone type for Java function clones as measured by both benchmarks in Table 8. We consider the benchmarks to agree if their measurements have an absolute difference no greater than 15%. This is a threshold we have used when comparing benchmarks in previous work [13]. We find that the benchmarks agree in 19 out of 27 cases, 70%. We now investigate the specific cases of disagreement.

Both benchmarks agree on the Type-3 recall of the tools which do formally support Type-3 detection, but disagree

TABLE 7: Mutation and Injection Framework vs. Bellon's Benchmark

	CCFX			ConQat			CPD			CtComp.			Deckard			iClones			NiCad			SimCad			Simian		
Java	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**
MF	98	77	0	91	90	86	99	74	0	96	48	0	-	-	-	100	93	96	100	100	100	100	96	89	81	90	0
Bellon	49	50	7	75	23	12	91	85	14	63	34	3	-	-	-	89	41	13	67	54	63	68	46	18	83	23	1
Agree?	○	○	⊗	○	○	○	⊗	⊗	⊗	○	⊗	⊗	-	-	-	⊗	○	○	○	○	○	○	○	○	⊗	○	⊗

C	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**
MF	99	82	0	-	-	-	98	0	0	69	40	0	73	73	69	100	96	99	99	99	99	100	97	89	85	97	0
Bellon	64	45	6	-	-	-	73	11	3	21	22	10	32	2	1	95	56	27	33	30	28	38	29	28	64	15	3
Agree?	○	○	⊗	-	-	-	○	⊗	⊗	○	○	⊗	○	○	○	⊗	○	○	○	○	○	○	○	○	○	○	⊗

TABLE 8: Mutation and Injection Framework vs. BigCloneBench

	CCFX			ConQat			CPD			CtComp.			iClones			NiCad			SimCad			Simian			Sourc.CC		
Clone Type	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**
MF	98	77	0	91	90	86	99	74	0	96	48	0	100	93	96	100	100	100	100	96	89	81	90	0	100	100	100
BCB	100	93	62	67	90	73	100	94	71	95	78	59	100	82	82	100	100	100	100	98	91	95	78	53	100	98	93
Agree?	⊗	○	○	○	⊗	⊗	⊗	○	○	⊗	○	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗

on the Type-3 recall of the tools which only offer Type-2 clone detection, including: CCFinderX, CPD, CtCompare and Simian. In all of these cases, BigCloneBench is measuring a significant Type-3 recall (53-71%), while the Mutation Framework is measuring no recall (0%). Neither benchmark is incorrect in these cases, but rather is telling us something different about the tools, and shows the need for both synthetic and real-world benchmarks.

Since the Mutation Framework requires the tool to detect the Type-3 features (mutations) in the Type-3 reference clones, it correctly measures 0% recall for the tools that do not support the detection of clones with gaps. In contrast, BigCloneBench only requires the tool to subsume 70% of the reference clones. It is showing that although these tools do not formally support Type-3 detection, they are able to report significant Type-1 or Type-2 regions within these very-strongly Type-3 clones. This is probably sufficient when a developer manually examines these clones and recognizes the Type-3 features outside of the reported clone boundaries. These tools would not be appropriate for automated clone analysis tasks as an automated clone analyzer would not be able to recognized the larger Type-3 clone the tool missed. BigCloneBench shows that the ability of the Type-2 clone detectors to locate significant cloned regions within the Type-3 clones significantly drops when the Type-3 clone similarity is less than 90% [17].

The benchmarks disagree on the Type-2 recall of CCFinderX, CPD and CtCompare, with the Mutation Framework measuring lower recall. These are cases where the Mutation Framework found particular weaknesses in these tools' Type-2 detection (Table **??**). CCFinderX struggled with Type-2 clones with arbitrarily renamed identifiers, CPD struggled with Type-2 clones with changes in numeric literal values, and CtCompare did not support the detection of Type-2 clones with either changes in numeric or string literal values. These tools have 90-99% Type-2 recall considering only the Type-2 mutations operators they do support, which agrees with the BigCloneBench results. The Mutation Framework measures average Type-2 recall as an equal weighted average across the Type-2 per-mutation-operator recalls, so poor performance for just one Type-2 mutation operator can drag down the average performance. It is possible these tools have much higher Type-2 recall with BigCloneBench because the kinds of Type-2 edits they do support are the ones most common in the BigCloneBench clones. While the Mutation Framework pinpoints the weaknesses and omissions of these tools' Type-2 detection, BigCloneBench suggests these tools may still perform well in the general case.

The benchmarks disagree on ConQat's Type-1 recall, with BigCloneBench measuring a significantly lower recall. In this case, ConQat was missing a significant number of small Type-1 clones of a particular functionality in BigCloneBench. We tried re-configuring ConQat for these clones and were unable to get it to detect them, suggesting a bug. In contrast, the Mutation Framework does not find any weakness in ConQat's Type-1 performance, despite evaluating it for every kind of Type-1 difference across a wide variety of syntax. Specifically, ConQat has problems with a large group of similar Type-1 clones found in IJaDataset, but the Mutation Framework shows it does not have problems in the general-case for Type-1 clone detection. The advantage of synthetic benchmarking is its comprehensive and exhaustive evaluation, while real-world benchmarks are limited to the variety of clones they contain. Still, the real-world benchmark demonstrates that clone detectors can perform well in synthetic tests but then encounter problems with a particular clone detection target.

Overall, the benchmarks agree for most recall measurements. Ignoring the cases where the tools do not formally support a clone type, the benchmarks agree in 89% (8 of 9) of the Type-1 cases, 67% (6 of 9) of Type-2 cases, and 100% (5 of 5) of Type-3 cases. Of the eight cases of disagreement, four (50%) were due to BigCloneBench not rejecting Type-1/Type-2 clones as sufficient capture of a Type-3 clone, there (38%) are cases where BigCloneBench measures higher Type-2 recall possibly due to differences in Type-2 edit distributions in real-world datasets (versus flat distribution in our synthetic benchmarking), and only one is an anomaly which we manually investigated (ConQat Type-1 detection).

The benchmarks have strong agreement, and most of the cases of disagreement are expected due to the differences in synthetic versus real-world benchmarking. We did not find any significant anomalies when comparing the benchmarks,

which builds our confidence in their accuracy.

5.3.4 Summary

In this section, we built our confidence in recall measurements by the Mutation and Injection Framework by comparing its results against our expectations for the clone detection tools and results from two real-world clone benchmarks.

We found strong agreement between our benchmark and our expectations of the tools, 89% for Java and 81% for C. Based on our knowledge of the tools, we do not find any concerning anomalies in the Mutation Framework results. Poor agreements between the Mutation Framework and Bellon's benchmark were consistent with some recent studies [13], [14], [15] which again show high confidence on our Mutation framework over Bellon's benchmark.

Next we compared the Mutation Framework against BigCloneBench and found strong agreement between the benchmarks, 70% for Java clones. Most of the disagreement was for cases where a clone detector had a higher recall with BigCloneBench for clone types that they do not formally support. Unlike the Mutation Framework, BigCloneBench cannot reject matches where the clone detector fails to capture the Type-2 or Type-3 elements of a reference clone. More precise evaluation is one of the advantages of synthetic benchmarking with the Mutation Framework. Ignoring these cases, the benchmarks agreed on 89% of the measurements.

Strong agreement between the Mutation Framework, BigCloneBench and our expectations, and the absence of any significant anomalies in the results, builds confidence that the Mutation Framework is an accurate benchmark. While Bellon's experiment was a significant triumph in measuring the relative performance of the contemporary clone detection tools [11], researchers agree that it may not be appropriate as a reusable benchmarking dataset [13], [14], [15], [40] for evaluating modern tools.

6 Extending the Framework - Case Study: Large Gap Clones

The Mutation Framework is user-extensible by allowing pluggable custom mutation operators. In this way the users can extend the framework to conduct recall evaluation experiments not covered by the default mutation operators. For example, with Wang et al. [46], we extended the Mutation Framework to evaluate the recall of clone detection tools for gapped clones.

Gap clones are a kind of Type-3 clone where the code fragments are identical except for a dissimilar gap due to the insertion of code into one of the code fragments. With Wang et al. [46], we introduced a new clone detection tool called CCAligner which aims to detect the large gapped clones that are outside the detection capabilities of standard Type-3 clone detection tools. To demonstrate the effectiveness of our tool over the existing Type-3 clone detectors, we used the Mutation Framework to synthesize gapped clones with various gap sizes.

To extend the Mutation Framework, we created a new gapped clone mutation operator. This operator takes a code fragment as input and outputs a new version of the code fragment with a block of contiguous source lines randomly inserted within. The operator can be configured for different gap sizes (number of source lines) so that recall can be measured for increasing gap sizes. An example of a clone generated by this mutation operator is shown in Figure 9, where a 6 line gap has been inserted. The operator includes a dataset of source lines selected from a large collection of open-source Java projects. It synthesizes a gap by randomly selecting source lines from this dataset, and inserts this block of code at a random line in the input code fragment.

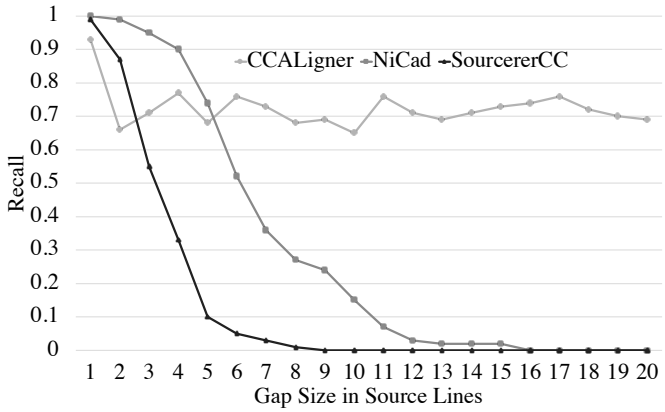

Fig. 5: Mutation Framework - Recall for Gap Clones for Various Gap Sizes

We used this mutation operator in an experiment measuring the recall of CCAligner for clones with gaps ranging from one to twenty lines. We configured the framework to generate 200 synthetic gapped clones per gap size, creating a benchmark with 2,000 synthetic gapped clones. We also evaluated NiCad and SourcererCC using this benchmark, as they are the top performing Type-3 clone detectors [45]. Recall for these tools is plotted by increasing gap size in Figure 5. As can be seen, while NiCad and SourcererCC have better recall for clones with a small gap, only CCAligner is able to maintain its recall for larger gaps.

It is very challenging to evaluate tools that target specific kinds of clones or domain specific clones as there may not be existing clone benchmarks that specifically highlight these clones. While benchmarks exist for Type-3 clones [11], [16], [18], none of the existing benchmarks specifically target clones with large gaps. Since large-gap clone research is still relatively new, building a high quality real-world benchmark would be very challenging as this kind of clone is not yet well understood. Extending the Mutation Framework to build a synthetic benchmark of large-gap clones was a great way to demonstrate that CCAligner is indeed detecting a classification of clones not found by the best of the existing tools, without requiring the extensive manual validation efforts a new real-world benchmark would require.

As large gap clones become better understood, a more diverse set of large-gap clone mutation operators can be created to better evaluate the tools. Once the importance of large-gap clones becomes clear, the investment to build a real-world benchmark of large-gap clones to compliment synthetic benchmarking will be warranted. With this case study we have demonstrated that the Mutation Framework

Line#	Original Fragment	Line#	Mutant Gapped Fragment
1	public BAD_PARAM invalidTaggedProfile(CompletionStatus cs, Throwable t)	1	public BAD_PARAM invalidTaggedProfile(CompletionStatus cs, Throwable t)
2	{	2	{
3	BAD_PARAM exc = new BAD_PARAM(INVALID_TAGGED_PROFILE, cs) ;	3	BAD_PARAM exc = new BAD_PARAM(INVALID_TAGGED_PROFILE, cs) ;
4	if (t != null)	4	if (t != null)
5	exc.initCause(t) ;	5	exc.initCause(t) ;
		6	HardObjectReference ref = HardObjectReference.peekPersisted(trans, id, 1);
		7	((ComponentUI) (uis.elementAt(0))).getAccessibleChildrenCount(a);
		8	oidStore.put(s.getName(), s) ;
		9	column = (column / TabInc * TabInc) + TabInc;
		10	dummyAction.removePropertyChangeListener(arg0);
		11	factories.add(new PrintStreamProviderFactory(ps));
6	if (logger.isLoggable(Level.WARNING))	13	if (logger.isLoggable(Level.WARNING))
7	{	14	{
8	Object[] parameters = null ;	15	Object[] parameters = null ;
9	doLog(Level.WARNING, "ORBUTIL.invalidTaggedProfile",	16	doLog(Level.WARNING, "ORBUTIL.invalidTaggedProfile",
10	parameters, ORBUtilSystemException.class, exc) ;	17	parameters, ORBUtilSystemException.class, exc) ;
11	}	18	}
12	return exc ;	19	return exc ;
13	}	20	}

TABLE 9: Example Gapped Clone Produced by the Mutation Framework

is a great benchmarking tool for evaluating domain-specific clone detectors even in the early stages of their research and development due to the low investment in manual efforts. Similarly, one could possibly write mutation rules for another domain such as finding API usability patterns or concept locations and could evaluate tools of that domain with little adaptations of the framework. Our framework was even adapted for working with models by Stephan and Cordy [52] where they evaluate model clone detection tools.

7 Threats to the Validity

A threat with the Mutation Framework is the synthetic clones may not be ones a real developer would create. The code fragments we randomly select for clone synthesis may not be ones a real developer would choose to clone. While the empirically validated clone editing taxonomy [6] guarantees our random mutations correspond to the types of edits real developers make on cloned code, it does not guarantee they apply edits a real developer would apply to the target code fragment. As well, since we focus on first-order mutations, the clones we generate may be simple compared to real-world clones. The advantage of synthetic clones is we can measure recall of the tools very precisely for each type of edit in the taxonomy. In our evaluation of our benchmark, we showed that synthetic benchmarking does yield unique insights into the tools not possible with previous benchmarks. However, we now have options in the framework where one can give a representative set of real clone pairs in the framework as input along with a subject codebase. The framework would then use those clone pairs for generating thousands of mutant codebases for evaluating the subject clone detection tools. This addresses the concern of synthetic clones to a great extent. It is also possible to apply higher-order mutations (applying more than one mutation operators) with minor changes in the framework which may more realistically reflect developers edit operations. Furthermore, we can overcome limitations in synthetic benchmarking with the Mutation Framework by pairing it with a real-world clone benchmark such as our BigCloneBench.

The current version of the Mutation Framework does not support the detection capability of Type-4 clones well. Given that most of the modern tools detect clones up to clones of Type-3, our focus in this work was to create a benchmark for clones up to Type-3. However, since our framework is extensible, future work on semantic transformations could be integrated into the framework as mutation operators. This is however challenging since semantic transformation is undecidable in general. In particular, one would need to create mutation operator that transforms a code fragment into a semantically equivalent but syntactically dissimilar version. It is also challenging to obtain a comprehensive taxonomy of edit operations for a given programming language for such semantic equivalency of code fragments.

While the Mutation Framework can generate very large benchmark corpora, the execution speed of subject clone detectors under evaluation is a limiting factor in the benchmark size. The tools need to be executed for each mutant system in the synthesized benchmark. While this is not a problem for most clone detectors which have execution times on the order of seconds for an average subject system, some clone detectors use more expensive algorithms and may require days of execution time to complete the benchmark. We generated benchmark corpora as large as possible considering the slowest of the participating clone detectors. Required execution time could be reduced by using a very small subject software system, or even a toy system of only two source files, but this harms the realism of the benchmark. While we could have reduced execution time by injecting all of the clones into a single software system, for any reasonably sized subject system the injected cloned code would overwhelm the original code and that this would violate the mutation analysis procedure. Nonetheless, with minor adaptations one could use this option in the mutation framework and reduce the execution time significantly.

We designed the framework to support as many clone detection tools as possible. We generate clones with strict block and function boundaries, which most tools support. The framework could be adapted to generate clones at arbitrary granularities, but these benchmarks would be incompatible with many tools. We generate clones whose boundaries can be reported by start and end source lines as most or all tools can report clones in this way. While some clone detection tools can report clones more precisely by start and end token or start and end character position, we restrict to line level boundaries for compatibility while evaluating them in the framework.

The framework guarantees that the mutated and injected code is syntactically valid, but does not guarantee that the modified source files will compile. Therefore, clone detectors that rely on compiled code may not be compatible with the framework. This is not a limitation in the Mutation Framework concept or procedure, but its current implemen-

tation. The framework could be made compatible by adding a repair process which fixes compile errors after clone injection with additional code injection and modification. This would be very challenging, and was not considered during implementation of the framework as very few available clone detectors require compilable code.

During the generation process the framework can constrain the synthesized clones with a minimum similarity threshold measured by line and/or token after source normalizations. This constraint was included to help the user configure their subject clone detection tools appropriately for the generated corpus. A limitation here is that not all tools use line-based or token-based similarity metrics, and even those that do may measure similarity differently. Therefore, the user may still need to experiment with thresholds to find appropriately configurations for their subject tools. The limitation could be overcome by augmenting the framework with additional similarity metrics, including variations on line-based and token-based measurements. However, this is not a major limitation in the framework as clones can be reliably generated with this constraint disabled.

The Mutation Framework does not identify cases where tools detect benchmark clones as *split clones*, which is a known problem in clone detection. This is when a clone detector reports a single clone as multiple clones split by differences in the clone. For example, if we generated a Type-2 clone with a changed string value, a clone detector may report the code before the string change as one Type-1 clone and the code after the string change as a separate Type-1 clone. We intentionally reject such a detection of the clone as it shows the tool is unable to handle the injected Type-2 difference. The Type-1 mutation operators will already show the tool is able to detect Type-1 clones, so with the Type-2 and Type-3 mutation operators we are measuring if the tool can detect such clones without splitting. While some detection algorithms may be prone to splitting, this can be corrected in post-processing by merging neighboring identical clones [53]. This is however not a limitation of the framework but is a limitation of the clone detection tools.

Our expectations of the tools' recall, which we used to evaluate the framework itself, may not be accurate. We formulated our expectations by consulting the tools' documentation, publications, literature surveys as well as their developers, where available. To account for some degree of inaccuracy, we allowed a $\pm 12.5\%$ range around the expectation. Where disagreement was found, we investigated if our expectation may be incorrect by also considering the BigCloneBench results and Bellon's benchmark.

Alternate tool configurations may result in better or worse performance in the tool evaluations. This is referred to as the confounding configuration choice problem by Wang et al. [44]. We took steps to ensure the tool configurations were appropriate for our study. We used configurations that target the known properties of the benchmark corpora, such as clone types, size and similarity thresholds. We consulted the default settings and documentation of the tools to choose these configurations. We were careful not to configure the tools in a way that would boost recall at a significant reduction in precision. This is the process an experienced user would use to configure these tools for their own subject system. Therefore, our results reflect what a user can expect from these tools. Furthermore, in this work, our primary objective was not to evaluate the clone detection tools in depth, rather to demonstrate that the Mutation Framework is a handy vehicle for evaluating and comparing the clone detection tools at a finer-granularity, in particular during the development process of the clone detection tools.

8 Conclusions

In this paper, we presented the Mutation and Injection Framework: an automatic evaluation framework for measuring the recall of clone detection tools. This framework uses an editing taxonomy for cloning to synthesize a reference corpus of artificial but realistic clones. The clone synthesis process mimics the copy, paste and modify cloning behavior performed by real developers. The framework enables a comprehensive reference corpus to be built without the need for manual candidate clone validation. The framework's capabilities extent to different clone types at a finer granularity (e.g., for different types of edit operations), two clone granularities (function and block) and three popular programming languages (Java, C, C#). The framework user has many controls over the properties of the generated reference corpus. The framework automates the execution and evaluation of subject tools for the reference corpus. It provides a full statistical report on the performance of the participating subject tools. The framework is controlled by a simple user interface, that allows users to control and share their experiments and reference clone corpora. The framework has also options where one could provide custom real world clone pairs as input along with a codebase for generating thousands of mutant codebases with those pairs for evaluating the tools.

As a demonstration of this framework, we used it to evaluate ten modern clone detection tools for the two clone granularities and three programming languages. We compared and evaluated these tools' recall at a very fine granularity. We compared these results against our expectations, and analyzed the cases of disagreement. Our findings suggest we can have confidence in the accuracy of our framework. In order to gain further confidence, we also compared the results from framework with two real-world benchmarks, our BigCloneBench and Bellon's benchmark. We found strong agreements with BigCloneBench which shows confidence of the mutation framework. We analyzed any disagreements, in particular with Bellon's in depth and showed that the the evaluation results of the Mutation Framework are consistent with the recent studies, again showing high confidence of the Mutation Framework. Overall, we found that synthetic and real-world benchmarks are highly complimentary, and the advantages of each allow us to fully explore clone-detection recall.

As future work, we plan to explore the use of higher-order mutations to produce more complex clones in a predictable and natural manner. We also plan to explore the creation of a comprehensive set of mutation operators for synthesizing Type-4 clones. We are also motivated to explore the extension of our mutation concept to the measurement of clone-detection precision, perhaps by injecting intentional

and common false-positive clones. The framework is available for others as open source at the first author's github page[1].

REFERENCES

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

[2] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in Proceedings of the 31st International Conference on Software Engineering, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 485–495. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070547

[3] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," Empirical Software Engineering, vol. 13, no. 6, p. 645, Jul 2008. [Online]. Available: https://doi.org/10.1007/s10664-008-9076-6

[4] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study," in 11th European Conference on Software Maintenance and Reengineering (CSMR'07), March 2007, pp. 81–90.

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

[6] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programing, vol. 74, no. 7, pp. 470–495, May 2009.

[7] 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.

[8] A. Charpentier, J.-R. Falleri, F. Morandat, E. Ben Hadj Yahia, and L. Réveillère, "Raters' reliability in clone benchmarks construction," Empirical Software Engineering, vol. 22, no. 1, pp. 235–258, 2017.

[9] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queens University, Tech Report TR 2007-541, 2007.

[10] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in WCRE, 2003, pp. 285–294.

[11] 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.

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

[13] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in The 30th International Conference on Software Maintenance and Evolution, ser. ICSME 2014, 2014, p. 10.

[14] B. Baker, "Finding clones with dup: Analysis of an experiment," IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 608–621, 2007.

[15] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, ser. EASE '15. New York, NY, USA: ACM, 2015, pp. 20:1–20:10. [Online]. Available: http://doi.acm.org/10.1145/2745802.2745821

[16] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 476–480. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.77

[17] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, Sept 2015, pp. 131–140.

[18] Jeffrey Svajlenko and Chanchal K. Roy and Farouq Al-Omari, "The Mutation and Injection Framework," https://github.com/jeffsvajlenko/MutationInjectionFramework, 2018.

[19] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in ICSE '08. ACM, 2008, pp. 321–330.

[20] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in ICSE '07. IEEE Computer Society, 2007, pp. 96–105.

[21] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in SAS '01, 2001, pp. 40–56.

[22] J. Krinke, "Identifying similar code with program dependence graphs," in WCRE'01, 2001, pp. 301–309.

[23] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," Software Engineering, IEEE Transactions on, vol. 32, no. 3, pp. 176–192, March 2006.

[24] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on, ser. SCAM'02, 2002, pp. 36–43.

[25] 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.

[26] F. Van Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in ASE'04, Sept 2004, pp. 336–339.

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

[28] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in WCRE, 1997, pp. 44–54.

[29] C. Roy and J. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, June 2008, pp. 172–181.

[30] T. Lavoie and E. Merlo, "Automated type-3 clone oracle using levenshtein metric," in Proceedings of the 5th International Workshop on Software Clones, ser. IWSC '11. New York, NY, USA: ACM, 2011, pp. 34–40. [Online]. Available: http://doi.acm.org/10.1145/1985404.1985411

[31] D. E. Krutz and W. Le, "A code clone oracle," in Proceedings of the 11th Working Conference on Mining Software Repositories, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 388–391. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597127

[32] Y. Yuki, Y. Higo, K. Hotta, and S. Kusumoto, "Generating clone references with less human subjectivity," in 2016 IEEE 24th International Conference on Program Comprehension (ICPC), May 2016, pp. 1–4.

[33] S. Schulze and D. Meyer, "On the robustness of clone detection to code obfuscation," in 2013 7th International Workshop on Software Clones (IWSC), May 2013, pp. 62–68.

[34] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," Empirical Software Engineering, vol. 23, no. 4, pp. 2464–2519, Aug 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9564-7

[35] ——, "Similarity of source code in the presence of pervasive modifications," in 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Oct 2016, pp. 117–126.

[36] S. Wagner, A. Abdulkhaleq, I. Bogicevic, J.-P. Ostberg, and J. Ramadani, "How are functionally similar code clones syntactically different? an empirical study and a benchmark," PeerJ Computer Science, vol. 2, p. e49, Mar. 2016. [Online]. Available: https://doi.org/10.7717/peerj-cs.49

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

[38] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in WCRE, 1997, pp. 44 –54.

[39] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in WCRE. IEEE Computer Society, 2003, pp. 285–294.

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

[41] C. Roy and J. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on, April 2009, pp. 157–166.

[42] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in Proceedings of

1. https://github.com/jeffsvajlenko/MutationInjectionFramework

the 7th International Workshop on Software Clones, ser. IWSC '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 8–9.

[43] J. Cordy, "The txl programming language," http://www.txl.ca/.

[44] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. ACM, 2013, pp. 455–465.

[45] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in Proceedings of the 38th International Conference on Software Engineering. ACM, 2016, pp. 1157–1168.

[46] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: A token based large-gap clone detector," in Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1066–1077. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180179

[47] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, Sept 2015, pp. 131–140.

[48] ——, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016.

[49] E. Kodhai and S. Kanmani, "Method-level code clone detection through lwh (light weight hybrid) approach," Journal of Software Engineering Research and Development, vol. 2, no. 1, p. 12, Oct 2014. [Online]. Available: https://doi.org/10.1186/s40411-014-0012-8

[50] F. Calefato, F. Lanubile, and T. Mallardo, "Function clone detection in web applications: A semiautomated approach," J. Web Eng., vol. 3, no. 1, pp. 3–21, May 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=2011138.2011140

[51] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in Proceedings of the 26th International Joint Conference on Artificial Intelligence, ser. IJCAI'17. AAAI Press, 2017, pp. 3034–3040. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172077.3172312

[52] M. Stephan and J. Cordy, "MuMonDE: A framework for evaluating model clone detectors using model mutation analysis," Software Testing, Verification & Reliability, p. 23 pp., 2018 (in press).

[53] N. Göde and R. Koschke, "Incremental clone detection," in Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on, March 2009, pp. 219–228.

Jeffrey Svajlenko Jeff Svajlenko received his Ph.D. from the University of Saskatchewan under the supervision of Chanchal K. Roy in 2018. He also holds a bachelor of science degree in computer science (with high honors) and the bachelor of science degree in Engineering: Engineering Physics (with great distinction) from the University of Saskatchewan. He is the recipient of NSERC CGSM, PGSD and PDF awards. His research interests include code clones, clone detection and benchmarking.

Chanchal K. Roy Chanchal K. Roy is an associate professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution including clone detection and analysis, program analysis, reverse engineering, empirical software engineering, and mining software repositories. He served or has been serving in the organizing or program committee of major software engineering conferences (e.g., ICSM, WCRE, ICPC, SCAM, ICSE-tool, CASCON, and IWSC). He has been a reviewer of major Computer Science journals including IEEE Transactions on Software Engineering, International Journal of Software Maintenance and Evolution, Science of Computer Programming, and Journal of Information and Software Technology. He received his Ph.D. at Queen's University, advised by James R. Cordy, in August 2009.