Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1662

Big data clone detection using classical detectors: an exploratory study

Jeffrey Svajlenko[1,*,†], Iman Keivanloo[2] and Chanchal K. Roy[1]

[1]*Department of Computer Science, University of Saskatchewan, Saskatoon, Canada*
[2]*Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada*

ABSTRACT

Big data analysis is an emerging research topic in various domains, and clone detection is no exception. The goal is to create ~~ultra-large~~ inter-project clone corpora across open-source or corporate-source code repositories. Such corpora can be used to study developer behavior and to reduce engineering costs by extracting globally duplicated efforts into new APIs and as a basis for code completion and API usage support. However, building scalable clone detection tools is challenging. It is often impractical to use existing state-of-the-art tools to analyze big data because the memory and execution time required exceed the average user's resources. Some tools have inherent limitations in their data structures and algorithms that prevent the analysis of big data even when extraordinary resources are available. These limitations are impossible to overcome if the source code of the tool is unavailable or if the user lacks the time or expertise to modify the tool without harming its performance or accuracy. In this research, we have investigated the use of our shuffling framework for scaling classical clone detection tools to big data. The framework achieves scalability on commodity hardware by partitioning the input dataset into subsets manageable by the tool and computing resources. A non-deterministic process is used to randomly 'shuffle' the contents of the dataset into a series of subsets. The tool is executed for each subset, and its output for each is merged into a single report. This approach does not require modification to the subject tools, allowing their individual strengths and precision to be captured at an acceptable loss of recall. In our study, we explored the performance and applicability of the framework for the ~~ultra-large~~ dataset, IJaDataset 2.0, which consists of 356 million lines of code from 25,000 open-source Java projects. We begin with a computationally inexpensive version of our framework based on pure random shuffling. This version was successful at scaling the tools to IJaDataset but required many subsets to achieve a desirable recall. Using our findings, we incrementally improved the framework to achieve a satisfactory recall using fewer resources. We investigated the use of efficient file tracking and file-similarity heuristics to bias the shuffling algorithm toward subsets of the dataset that contain undetected clone pairs. These changes were successful in improving the recall performance of the framework. Our study shows that the framework is able to achieve up to 90–95% of a tool's native recall using standard hardware.

Received 9 October 2013; Revised 20 March 2014; Accepted 28 May 2014

KEY WORDS: shuffling framework; clone detection; scalability; big data; clone corpus

1. INTRODUCTION

Scalable clone detection is among the most active topics in the clone community. One of its primary goals is the creation of clone corpora from ~~ultra-large~~ inter-project datasets that often contain in the order of thousands of open-source systems. However, building scalable tools is challenging, and it is often impossible to use existing state-of-the-art tools for big data analysis, except for emerging tools that are built for extreme scalability. Reasons for their failure include insufficient memory, impractical computation time, and/or limitations in their underlying algorithms.

*Correspondence to: Jeffrey Svajlenko, Department of Computer Science, University of Saskatchewan, Saskatoon, S7N 5C9, Canada.
†E-mail: jeff.svajlenko@usask.ca

In our research, we develop and evaluate a scalability heuristic that we call the shuffling framework [1, 2]. Our technique allows classical clone detection tools (i.e., those not specifically designed for big data) to be scaled to big data on standard workstation-class hardware without modification. The framework achieves scalability by executing the classical tool for subsets of the dataset. The subset size is kept small enough that the tool does not encounter scalability issues when executed on a standard workstation. The subsets are chosen by a non-deterministic process that 'shuffles' the dataset's files into inputs for the classical tool. Using the tools in their original state ensures that their native precision and detection characteristics are maintained when executed through the framework. By executing the tool for a sufficient number of subsets, an acceptable ratio of the tool's native recall is achieved for a dataset outside of its native scalability. The key to the performance of the framework is the design of the non-deterministic strategies used to choose the subsets.

This research is motivated by the richness of inter-project clone corpora for software mining experiments and applications. Inter-project datasets of interest include public open-source repositories (e.g., SourceForge and GitHub) as well as private corporate repositories. Clone corpora may be mined to study developer behavior both globally (e.g., across open-source repositories) and within an organization (e.g., across a company's private repository). They can be used to discover frequently re-implemented functionalities that should be extracted into new software libraries to remove duplicated engineering costs. A corpus may also be used as a basis for Internet-scale clone search [3], which has applications including API recommendation and usage support. Scalability in detection is achieved by either using novel scalable detection techniques (general or domain specific) or mixing classical approaches with scalability heuristics.

One of our goals is to allow classical tools to contribute toward inter-project clone corpora (e.g., [3]). It is not sufficient to only consult scalable clone detectors when creating a clone corpus as classical tools have their own unique strengths and detection characteristics. While general-purpose scalable detection techniques exist in the literature, most have not been publicly released as user-friendly tools. Additionally, scalable tools are still novel, and their recall and precision have not been proven. Classical tools have matured, and there is more understanding and confidence in their abilities and detection quality. In order to build a truly comprehensive inter-project clone corpus, a variety of detection tools need to be consulted, including both scalable and classical tools.

In our earlier research [1, 2]. we proposed a shuffling framework based on input partitioning. This strategy completely partitions the dataset into disjoint subsets. The tool is then executed for each subset to locate the clones within these partitions. Because it is likely that files containing clones will be assigned to different partitions, the contents of the partitions are randomly shuffled over a number of rounds. Rounds are executed until either the framework user has met their time constraints or the cost of executing an additional round exceeds the expected benefit (as judged from the previous rounds) in terms of the number of new clones detected. This strategy relies upon randomization to shuffle clones together. It assumes that while a large number of clones remain to be found, there is a good chance random selection will shuffle files containing clones into the same partitions. The technique should reach a point of diminishing returns when a significant portion of the tool's native recall has been found. This shuffling strategy is computationally cheap, and the creation of the subsets is negligible compared with the cost of executing the clone detection tool.

In this research, we evaluate this partition strategy for the ~~ultra-large~~ inter-project dataset IJaDataset 2.0 [4, 20] using a selection of classical clone detection tools, including Deckard [5], NiCad [6], iClones [7], Simian [8], SimCad [9], and CCFinderX [10]. We measure the framework's detection performance as the ratio of a tool's native recall that it is able to capture. This study reports our observations and the challenges faced in executing our framework for these tools and dataset. In order to gauge the expected performance of the framework for these tools, we also executed it for standard-size datasets, which allowed us to compare clone detection with and without the framework. We developed and evaluated a heuristic for estimating framework performance when the clone output was too large to process on available hardware.

From our performance observations, we identified the strengths and deficiencies of the partitioning approach. Generating the subsets by randomly partitioning the dataset over a number of shuffling rounds is computationally inexpensive and ensures the tool is exposed to every file in the dataset. However, we found that a large number of rounds were required to obtain an acceptable ratio of a tool's native recall. We identified two attributes of random partitioning that limits its performance.

DOI: 10.1002/smr

First, when shuffling the partitions, the framework does not consider which files have been shuffled together previously. It is possible, especially as more rounds are executed, that pairs of files will be randomly shuffled together more than once. Executing the tool for the same pairs of files repeatedly costs computation time and resources without discovering new clones (i.e., improving recall). Computing rounds of partitions that never shuffle the same files together more than once is neither simple nor cheap. However, minimizing the reshuffling of the same files together repeatedly would improve the performance of the framework.

Second, this strategy does not consider the similarity of the files it shuffles together. A significant portion of a clone detector's computation time is spent searching for clones between files that do not contain clones. Shuffling together only those files that contain clones would reduce the amount of wasted time. Of course, determining if two files contain a clone has the same cost as clone detection. However, a cheap (i.e., $O(n)$, where n is the combined length of the files) heuristic to estimate if two files contain enough similar code to *possibly* contain a clone could be used to prevent files too dissimilar to contain a clone (as judged by the tool) from being shuffled into the same subsets.

Using these observations, we improved our framework's subset generation and file shuffling strategy. Specifically, we explored methods of efficiently tracking seen file pairs and efficient heuristics to measure source file similarity. By tracking the seen file pairs, we can guarantee that each new subset contains a minimum number of new detection experiences. By measuring file similarity, we can avoid shuffling together files that are unlikely to contain a clone as judged by the specific classic tool. The goal of these two heuristics is to maximize the number of clones found per subset the tool is executed for, which minimizes the number of subsets needed to obtain an acceptable ratio of the tool's native recall. This improves the scalability of our framework. We incrementally introduced these heuristics to the framework and measured their performance in an experiment mimicking a real big data scenario.

Using our findings of the computational cost and recall performance of the added heuristics, we specified a final shuffling algorithm that merged the best features of the partitioning method and the heuristics. We used this final version of the shuffling algorithm to analyze the IJaDataset. We compared the improved algorithm against the original core algorithm (the original shuffling framework [1, 2]). While the heuristics increased the cost of generating the subsets of the dataset to analyze, it greatly reduced the number of subsets the classical tool needed to be executed for it to achieve a satisfactory ratio of its native recall, while of course retaining the tool's original precision.

In summary, this work answers the following research questions:

RQ#1 What is the accuracy of our heuristic for measuring the recall performance of the shuffling framework?

RQ#2 What is the expected recall performance of the core shuffling framework for these selected clone detection tools with respect to their native recall performance?

RQ#3 Is our shuffling framework successful in scaling ~~classical detection~~ tools to big data?

RQ#4 By observing the behavior of the shuffling framework, can we modify it to improve its ~~recall~~ performance in terms of recall and execution time?

RQ#5 Does the improved shuffling framework perform better for big data?

We begin with a short survey of related work in Section 2. Section 3 provides essential background information, including key definitions. The procedure of our core shuffling framework ~~as~~ proposed in previous work [1, 2] ~~is~~ outlined in Section 4. Section 5 overviews our experimental setup and defines our metrics, including the recall evaluation heuristic (RQ#1). We evaluate the expected performance of our core algorithm (RQ#2) in the preliminary experiments detailed in Section 6. Section 7 discusses our experiences in applying our core shuffling framework to big data (IJaDataset) and reports our observations regarding the framework's clone detection performance (RQ#3). In Section 8, we analyze the performance and deficiencies of our core shuffling algorithm. In Section 9, we develop shuffling heuristics to address the deficiencies in the core approach and incrementally integrate them into the core shuffling algorithm. Using a test dataset (a sample of the IJaDataset), we measure the effectiveness of these improvements versus the costs the heuristics added to the shuffling algorithm.

DOI: 10.1002/smr

From these experiments, we specify an improved shuffling framework in Section 10. In Section 11, we revisit the IJaDataset with the improved framework and compare our experiences against the core framework in terms of recall performance and tool scalability improvements (RQ#5). We conclude our research in Section 12 and outline our future work in Section 13.

2. RELATED WORK

Scalable clone detection research can be summarized as five unique approaches: (1) deterministic novel general-purpose detection (e.g., [11]); (2) deterministic novel domain-specific approaches (e.g., [12]); (3) deterministic approaches for achieving scalability by altering available tools (e.g., [13]); (4) deterministic approaches for achieving scalability using an available clone detection tool as is (e.g., [14]); and (5) non-deterministic approaches for scaling an available tool (e.g., [1]). A variety of use cases can be addressed using each family based on their unique features. Our shuffling framework is an implementation of approach (5).

Deterministic novel general-purpose approaches, (1), are designed specifically for scalability. A number of such techniques for big data have been explored in clone literature; however, public tool availability remains rare. Approaches that achieve scalability on a single machine may require compromises in granularity, recall, and/or precision in order to reduce computational complexity or clone search space. Other approaches achieve scalability by targeting scalable hardware, such as cloud-based computing clusters, which can be costly to purchase or rent.

Deterministic novel domain-specific approaches, (2), achieve scalability by optimizing for a particular use case. By exploiting domain knowledge of a particular use case, computational complexity can be lowered without significant compromise to detection features, recall, or precision. However, the approach's performance is strongly specific to its domain of study. The approach may be ineffective in other use cases.

Existing classical tools may be modified for scalability, (3). These approaches exploit the proven existing clone detection technique with modifications to improve its scalability. For example, an existing tool may be modified to distribute its computation. Or a heuristic may be used to reduce the search space the classical approach must be executed for. Improving the scalability of an existing tool may scale its hardware requirements or reduce its recall and/or precision. To implement such an approach, the original tool's source code and expert knowledge of its implementation are required. Many tools are closed source or are released without extensive design documentation.

Methods (4) and (5) use classical tools as is and scale them to big data. These approaches exploit the known detection characteristics, recall, and precision of available tools. Many classical tools are available, and users are confident in their abilities and correctness because of their widespread use in clone research. By not requiring modification to the tools, these approaches can scale closed-source tools. While open-source tools might be modifiable for increased scalability, this requires expert knowledge in their algorithms and implementations. Our shuffling framework exploits the non-deterministic method, (5).

Deterministic methods of scaling classical tools without modification, (4), are the most similar approach to our shuffling framework. An implementation of the deterministic approach (e.g., [14]) begins by partitioning the input dataset into disjoint subsets half the size manageable by the classical tool on workstation hardware. The tool is then executed for each pair of these subsets. If the tool's input scalability limit is $1/n^{\text{th}}$ of the big dataset, then the deterministic method will partition the input into $2n$ disjoint subsets and execute the tool for $n(2n-1) = O(n^2)$ subset pairs. This strategy maintains a classical tool's native recall and precision while scaling it to big data by deterministic exposure of the tool to every pair of files in the dataset. However, for a dataset containing thousands of software systems, the deterministic method may require several weeks of execution time with a classical tool. Therefore, to achieve scalability in execution time, the user must distribute the analysis of the subset pairs across a cluster of workstations.

Our shuffling framework aims to scale classical tools using a single workstation, or a (small) handful of available workstations, without modifications to the tool. It executes the tool for some number of manageable subsets of the dataset. The subsets are chosen by a non-deterministic (random) process,

DOI: 10.1002/smr

meaning that the tool is not exposed to every file pair in the dataset. The approach relies upon randomization to allow an acceptable ratio of a tool's native recall to be achieved in a manageable number of subsets, at least far fewer than required by the deterministic method (i.e., approach (4)). The probability of files containing clones ending up in the same subset is higher when a large ratio of a tool's native recall remains to be found. This approach maintains the classical tool's detection characteristics and precision but sacrifices its recall to achieve scalability on commodity hardware. In this research, we develop and investigate non-deterministic methods of choosing these subsets so as to reduce the number of subsets needed to achieve an acceptable ratio of a tool's native recall.

There are a few recent and similar studies to our research. Ishihara *et al*. [12] exploited inter-project scalable clone detection to locate commonly used functionalities within 13K open-source projects in order to generate a seed for future APIs and libraries. Schwarz *et al*. [15] studied cloning between 3K Smalltalk projects to deploy a database of clones that can be queried. Ossher *et al*. [16] observed cloning at the file level using coarse-grained clone detection heuristics. Common to all these studies, the detection approach is customized and optimized considering the research objectives and requirements. This is contrary to our research where we show that a clone dataset can be generated using available clone detection tools by coping with the scalability issue without altering the tools.

3. BACKGROUND

Similar source code within a software system, or some other collection of source code, are called code clones or software clones. Developers and researchers are interested in clones that are similar either textually, syntactically or functionally. The most common source of clones is copy-and-paste code reuse. However, clones may arise from a number of other developer actions [17]. Clone detection tools are used to locate clones in software systems. Clones are frequently reported as pairs of similar code fragments, which can be summarized into clone classes.

Code fragment A continuous slice of source code in a single source file, specified by the triple (source file, start line, end line).

Clone A pair of code fragments that are considered similar by some definition of similarity, also called a *clone pair*.

Clone Class A set of code fragments that are considered similar by some definition of similarity. Each code fragment in the set forms a clone pair with each of the other code fragments. For example, a clone class of size 5 summarizes 10 distinct clone pairs.

Clones are frequently assigned a clone type that describes the nature of the similarity between its code fragments. Researchers agree upon four fundamental clone types [18, 19].

Type 1 Code fragments that are syntactically identical with the exception of differences in comments, white space, and layout.

Type 2 Code fragments that are syntactically identical with the exception of differences in identifier names, literal values, comments, white space, and layout.

Type 3 Code fragments that are syntactically similar with differences occurring at the statement level. Code fragments may have statements added, removed, or changed with respect to one another.

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

Type 3 and 4 clones lack precise definitions. Researchers do not agree upon the maximum dissimilarity allowed between type 3 clones, or even how this dissimilarity should be measured [18]. Similarly, functional similarity is not well defined, and type 4 should probably be split into multiple types of functional similarity. Currently, very few tools consider functional similarity. For this reason, we focus only on the first three clone types in this work.

Clone detection performance is measured using the information retrieval metrics recall and precision. Recall measures the tool's proficiency at locating and reporting true-positive clones, while precision measures its detection accuracy.

DOI: 10.1002/smr

Recall The ratio of the clones within a software system or repository that a tool is able to detect.
Precision The ratio of the clones detected by a tool that are true clones and not false positives.

4. THE CORE SHUFFLING FRAMEWORK

The shuffling framework allows clone detection tools not designed for extreme scalability to scale to ~~ultra-large~~ datasets without modification on standard hardware while achieving an acceptable overall recall and retaining the tool's native precision. Summarized in the following is our core shuffling framework approach as proposed in our earlier work [1, 2]. We begin this research with the analysis of the core approach's performance for ~~ultra-large~~ datasets. We discuss improvements to this approach starting in Section 10. The core framework executes the following procedure:

1. The source files of the dataset are randomly partitioned into n disjoint subsets of equal size. Subset size is chosen such that the clone detection tool can handle a single subset within a single execution on standard hardware without encountering scalability difficulties.
2. Each subset is searched independently by the clone detection tool. This can be performed serially, in parallel, or in a distributed fashion over independent computers.
3. The detected clone pairs are merged into a clone repository.
4. Steps (1) through (3) are repeated for r rounds. Multiple rounds are required as a single round achieves limited recall. There is a high chance that cloned contents are assigned to disjoint subsets. ~~Because~~ the rounds are independent, they may be executed serially or in parallel on common or independent computing resources.

The framework achieves scalability by partitioning the dataset into subsets individually manageable by the clone detection tool. The tool's recall is recovered by repeating detection after shuffling the partition contents. The goal of this non-deterministic approach is to achieve an acceptable fraction of the tool's native recall within a manageable number of rounds ($O(nr)$ tool executions).

To use this framework, the user must select an appropriate subset size for their clone detection tool. Factors affecting this choice include how the tool's memory requirements, computation time, and algorithmic complexity scale with input size. Some tools may also have inherent input size limitations in their algorithms and data structures.

A number of rounds to execute must also be chosen. The more rounds executed, the closer the framework will come to the tool's native recall. However, the number of rounds executed must be manageable within available time and computing resource constraints. Preferably, rounds should be executed until the number of new clones found (i.e., discovered in the most previous rounds) is no longer worth the additional computation time. This decision depends on the individual use case.

Clones detected in each subset are added to a single clone repository. A clone may be detected in multiple rounds if its files are randomly shuffled into the same partitions in multiple rounds. ~~Therefore~~ the clone repository must handle the insertion of duplicate clones by retaining only one copy of the clone. In our implementation of the framework we used a hash-based set as a clone repository. This provides amortized O(1) clone insertion and lookup. Our clone pair equivalence function is defined to ignore code fragment order, so the set will only retain one copy of a clone pair even if the code fragment order is reversed.

5. STUDY SETUP—THE CORPUS, ENVIRONMENT, TOOLS, AND MEASURES

5.1. *Corpus—IJaDataset 2.0*

For our experiment, we used the second version of the IJaDataset, which was constructed using raw data crawled in 2012 [4]. The dataset covers the source code of approximately 25,000 open-source Java projects. This new version of the dataset contains up-to-date source code and is two times larger than the first version, which we used in our earlier studies [1]. The dataset is based on source

DOI: 10.1002/smr

files mined from SourceForge and Google Code in 2012. The dataset includes nearly 3 million Java source files spanning 356 million lines of code (LOC). The dataset is publicly available [20].

Of the three million files in the IJaDataset, 6238 are greater than 2000 lines in length. While these make up an insignificant portion of the dataset, they may contribute considerably to a clone detection tool's execution time. For this reason, we consider these files as outliers of the dataset and omitted them from the experiments.

5.2. Hardware

Our framework aims to scale classical tools to ~~ultra-large~~ datasets using *standard hardware*. Clone detection in big data is mostly of interest to researchers and professional developers. For this reason, we used consumer-grade workstation-class desktop computers as our target for standard hardware. We expect such machines to have four or more processing threads on a modern CPU architecture (e.g., Intel Core i5) and 8–32 GB of system memory. These machines should store active data on either a performance hard drive or, ideally, a solid-state drive. At the time of publication, machines meeting these specifications cost approximately $US800–1500.

The first IJaDataset experiment (Section 7) was executed in a distributed fashion on computing instances provided by the Bugaboo cluster of the Western Canada Research Grid (WestGrid) and Amazon EC2. These instances meet our definition of standard hardware, and multiple instances were exploited in order to complete this study in a limited time frame. The average instance included a 2.66 GHz quad-core processor, 12 GB of memory, and two 10,000 RPM hard drives in RAID0. All other experiments were executed on our local hardware, which includes a 3.6 GHz quad-core processor, 16 GB of memory, and a single consumer-grade solid-state drive. For particularly demanding analysis of the experiment's results (e.g., gold dataset creation and performance measurement), an EC2 instance with 64 GB of memory was utilized. This extraordinary instance was never used for steps of the shuffling framework, only for analysis of the framework's performance.

Upon completion of the first IJaDataset experiment, we realized that traditional hard-disk drives are the bottleneck to the framework. Subset creation and clone detection were considerably faster on our local machine using a consumer-grade solid-state drive. For the experiments using standard hard-disk drives, we include estimates of what the execution time would have been on our local hardware based upon our findings with later experiments.

5.3. Clone detection tools

For this study, we explored six clone detection tools. Being freely available and supporting Java source code were our major deciding factors. Table I summarizes our selected tools and their chosen configurations. When possible, we preferred the tools' default settings. We used the same tool versions and configurations across all experiments.

5.4. Measures

The performance of our framework is measured as total recall: the ratio of the clones from the target tool's gold standard that the framework is able to find. The gold standard is the clones the target tool finds when run as is (i.e., without our framework). For the application of the shuffling framework for r rounds and n subsets, total recall is calculated using Eqn. 1.

$$tr(r,n) = \frac{\left|\left(\bigcup_{i=1}^{r}\left(\bigcup_{j=1}^{n} detected\ clone\ pairs(i,j)\right)\right) \cap (clone\ pairs\ in\ gold\ standard)\right|}{|(clone\ pairs\ in\ gold\ standard)|} \quad (1)$$

The numerator is the number of clone pairs in the tool's gold standard that it detected when executed through the framework. The set on the left of the set intersection is the set of unique clone pairs detected by the tool using the framework. The first union iterates over each round of the framework, while the second iterates over each subset in a round. *detected clone pairs*(i, j) is the set of clone pairs detected in subset j of round i. The denominator is the number of unique clones in the gold

DOI: 10.1002/smr

Table I. Tool configurations.

Classical subject tools	Configurations
Deckard [5] (version 1.2.3)	Minimum fragment size of 50 tokens and a sliding window of five tokens; minimum 90% clone similarity (tree-based metric)
NiCad [6] (version 3.4)	Normalized fragment size of 10–2500 lines and minimum 70% clone similarity (line-based metric)
iClones [7] (version 0.1.2)	Minimum clone fragment size of 100 tokens and minimum cloned block size of 20 tokens
Simian [8] (version 2.3.33)	Code fragment sizes of six lines or greater, no identifier or literal renaming
SimCad [9] (version 2.1)	Detection of clone pairs of all types after consistent identifier normalizer
CCFinderX [10] (version 10.2.7.4)	Minimum fragment size of 50 tokens, with a minimum unique token type of 12

standard. As this metric considers clone pairs, we also refer to it as clone recall or clone pair recall. It measures the ratio of the tool's native recall that the framework achieved.

We measure performance in terms of clone pairs instead of clone classes for a number of reasons. All clone detection tools either support clone pair output, or their clone class output can be simply converted into clone pairs. For tools that only report clone pairs, it is not trivial to convert their output to clone classes. It would require either modification to the tool or implementation of a clone-clustering algorithm that uses the same decision logic and clone metrics as the tool. Disabling clone clustering in tools where it is an option may also reduce their computation time and memory requirements. We disabled clone class output when possible to improve the native scalability of the tools.

Finally, considering clone pairs makes the calculation of total recall much more efficient. The clone reports from a tool executed by the framework can be merged into a hash set. The complexity of determining if a clone pair in the gold standard has been detected is then $O(1)$. ~~Because~~ the number of clones these tools detect in IJaDataset is very large, the $O(1)$ complexity is essential. This way, the evaluation of total recall is linear with respect to the size of the gold standard.

5.4.1. Heuristic-based total recall measurement. In our experience, a clone detector's output for big data datasets may be too large for the calculation of total recall in a reasonable time frame, even when extraordinary hardware is utilized (e.g., 244 GB of random access memory (RAM)). Specifically, we experienced this when attempting to measure total recall for the framework's evaluation of the IJaDataset using Simian. For this reason, a heuristic was devised to estimate total recall using limited time and resources. This heuristic estimates total recall by measuring the ratio of the cloned fragments, rather than clone pairs, from the gold standard that are found using the framework. Heuristic recall is measured using Eqn. 2. The notation is the same as in Eqn. 1, except for cloned code fragments. As this metric considers cloned fragments, we also ~~referred~~ to it as cloned fragment recall or fragment recall.

$$hr(r,n) = \frac{\left|\left(\bigcup_{i=1}^{r}\left(\bigcup_{j=1}^{n} detected\ cloned\ fragments(i,j)\right)\right) \cap (cloned\ fragments\ in\ \sout{gold})\right|}{|(cloned\ fragments\ in\ \sout{gold})|} \quad (2)$$

This heuristic is based on the assumption that if two cloned fragments of a clone pair have been found by our approach, then there is a good chance that the clone has also been detected or that the clone could be recovered by applying the transitive property to all found clone pairs. For example, if fragments f_1, f_2, and f_3 have been found in clone pairs (f_1, f_2) and (f_2, f_3), then the missed clone pair (f_1, f_3) can be recovered. A caveat of this approach is that while it holds true for all clones of types 1 and 2, it does not for all type 3 clones.

5.4.2. Evaluation of our heuristic-based recall measure. In this study, we tested the assumptions of our heuristic-based recall measurement. We searched JDK1.7 using NiCad, Simian, and Deckard both as they are and with our shuffling framework. The framework was parameterized to evaluate

DOI: 10.1002/smr

the dataset for 15 subsets over 30 rounds. Figure 1 compares the total recall and heuristic recall for the tools after each round. For NiCad and Simian, the transitive property was applied to recover additional clones. Recovered recall was then evaluated as in Eqn. 3 by including the recovered clones per round as part of the tool's detected clones. Recovered recall was not evaluated for Deckard because of the size of its output.

$$rr(r,n) = \frac{|((detected\ clone\ pairs) \cup (recovered\ clone\ pairs)) \cap (clone\ pairs\ in\ gold)|}{|clone\ pairs\ in\ gold\ standard|} \quad (3)$$

As can be seen from these experiments, heuristic recall overestimates the total recall but follows a similar trend. Both show logarithmic growth in recall across the rounds. Cloned fragment (heuristic) recall starts higher but has a slower growth across the rounds. The recovered recall performance for NiCad and Simian shows the correctness of the heuristic. For NiCad, the recovered recall approximately matches the heuristic recall. For Simian, the recovered recall approaches heuristic recall after half of the rounds have been executed. This shows us that our heuristic is effective in estimating the recall of our shuffling framework (RQ#1).

In this study we applied the transitive property naively. We assumed it held for all type 3 clones. This means we "recovered" false positive clones in the cases where transitivity did not hold between type 3 clones. However, these false positives do not affect the measure of recovered recall. Therefore, these results represent the ideal case where the recovery method successfully recovers all transitive clone pairs. In practice, a recovery technique would need to check that transitivity held before applying it to type 3 clones. It may not be possible to implement an efficient check that accepts all true positive transitive clones and rejects all false positive transitive clones. We used transitive clone recovery only in this evaluation of the heuristic recall measure. Creating an efficient

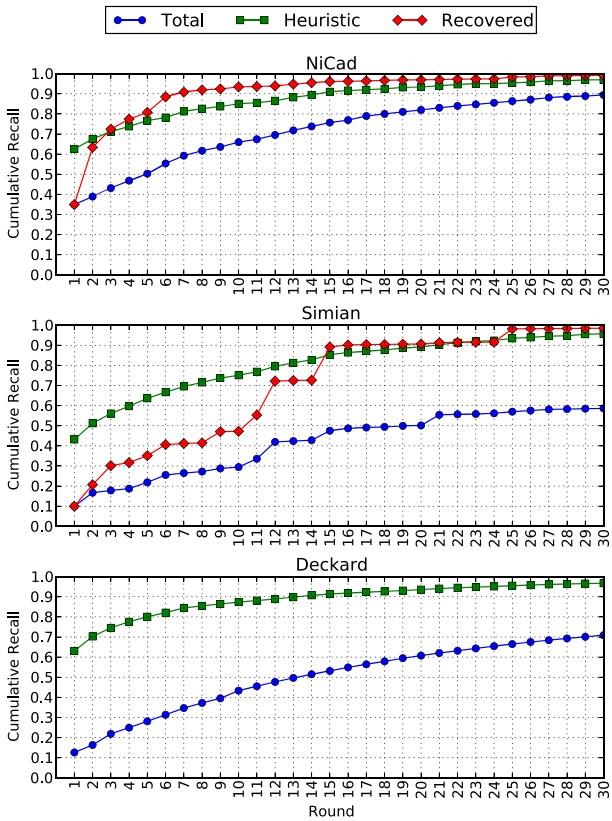


Figure 1. Comparison of the recall estimation approaches.

DOI: 10.1002/smr

transitive check with high recall (accepts most true positive transitive clones) and precision (rejects most false positive transitive clones) is a topic of future work. In this paper we use our total recall and heuristic recall measurements to comment on if a transitive clone recovery method is worth perusing in future work.

6. PRELIMINARY EXPERIMENTS

We used the shuffling framework to evaluate three regular-sized subject systems. This allowed us to evaluate the systems with the tools both natively (their gold standard) and with the framework. The goal of this experiment was to observe the expected performance of the framework for the six selected tools (RQ#2). We chose JHotDraw (20KLOC—285 files), ArgoUML (190KLOC—1845 files), and JDK1.7 (900KLOC—6916 files) as our regular-sized systems. The framework was parameterized for 15 random subsets and 30 detection rounds.

The framework's total recall performance for each tool's detection of JHotDraw54b1 is shown in Figure 2, that of ArgoUML in Figure 3, and that of JDK1.7 in Figure 4. The legends of these graphs specify the gold standard size (number of clones) for each tool. The framework performed

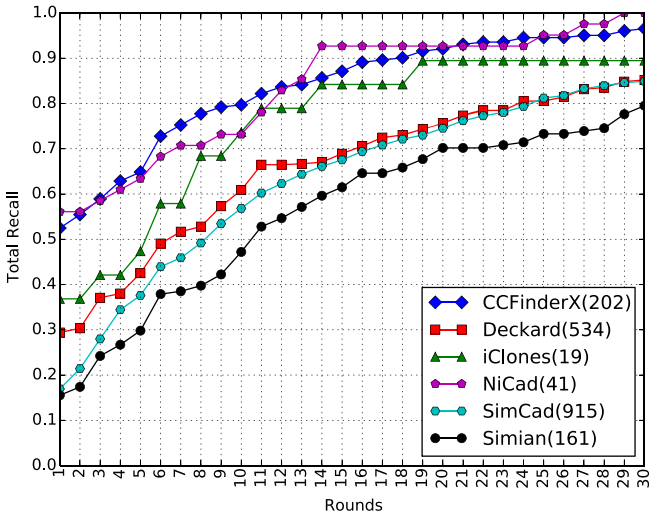


Figure 2. Preliminary experiment—JHotDraw54b1.

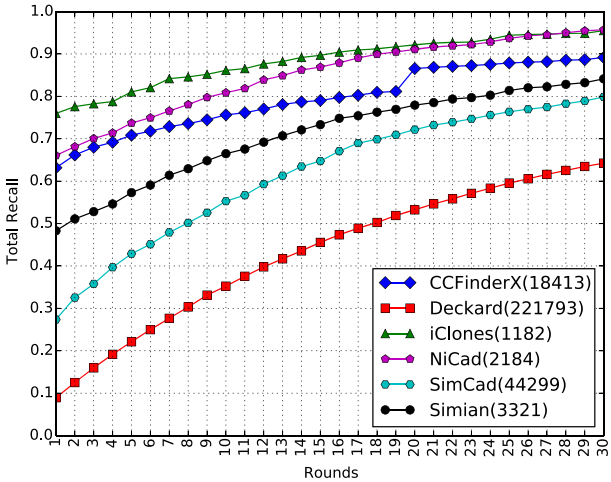


Figure 3. Preliminary experiment—ArgoUML.

DOI: 10.1002/smr

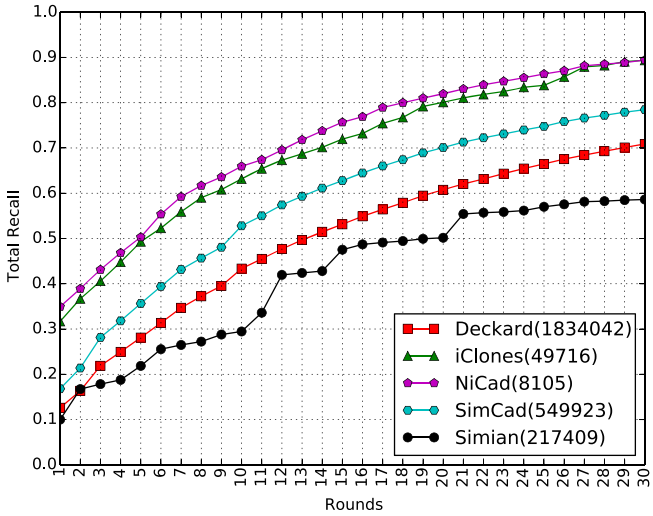


Figure 4. Preliminary experiment—JDK1.7.

very well with NiCad, iClones, and CCFinderX, obtaining a high total recall after 30 rounds. It struggled more for Deckard and performed poorly with Simian for JDK1.7. Total recall started and ended lower for JDK1.7 but increased faster than for ArgoUML and JHotDraw, likely because of the differences in the sizes of the two systems (and gold standards). CCFinderX is omitted from the JDK1.7 experiment because of failure during detection.

In all cases, we see approximately logarithmic growth in total recall across the rounds. As total recall becomes larger, the increase in total recall from each round decreases. This is expected, as the smaller the ratio of a tool's native recall that is left to be found, the lower the probability the files containing these undetected clones will be shuffled into the same subset. We saw the same trend with heuristic and recovered recall in the heuristic study (Section 5.4.2).

An observation from this experiment is that the larger the gold standard, the lower the total recall obtained by the framework across the same number of rounds and subsets. This is seen here for both variation in detection tools and subject system size. The exception being Simian, for which the framework achieves a lower total recall than for tools with larger gold standards. Perhaps Simian has better precision for smaller inputs and is therefore not finding the false positives in its gold standard, leading to a lowered total recall.

These results indicate that the framework can achieve an acceptable ratio (>70%) of a tool's native recall given an acceptable number of rounds. The plots here show the expected framework performance for the tools, which answers RQ#2.

7. MOTIVATING STUDY—IJADATASET

In this experiment, the clone detection tools were executed through the core shuffling framework to detect clones in IJaDataset 2.0. This experiment was used to evaluate the performance and feasibility of the shuffling framework for clone detection in big data using classical tools (RQ#3).

Using a rented Amazon EC2 instance with 64 GB of memory and 10,000 input/output operations per second, we were able to obtain Simian's gold standard for IJaDataset. This allowed us to compare native versus framework recall in a big data scenario. We were unable to obtain gold standards for the other tools. The required processing time and computer memory exceeded our available time and hardware rental budget. Simian is atypical in that it was scalable to big data within a reasonable execution time when a large amount of RAM was provided. However, Simian's detection capabilities are not as sophisticated as the other tools; for example, it only detects type 1 and ~~2~~ 1 clones.

Of the six selected tools, only Simian, NiCad, and Deckard were used successfully for this experiment. CCFinderX, iClones, and SimCad were omitted because of compatibility issues with the

DOI: 10.1002/smr

dataset. These tools terminated with an error message if a parsing error was encountered instead of skipping the offending file. ~~Because~~ we used the largest subset size these tools could handle, the chance of a parsing error in a single subset is very high. This caused the tools to make very little progress in a round of the shuffling framework, as they used execution time but produced no clone reports for a large number of the subsets. The omitted tools are further discussed in Section 7.4. Table II summarizes the shuffling experiments performed.

7.1. Simian

7.1.1. Setup. Simian was chosen for this experiment as it was possible to obtain its gold standard for IJaDataset. This allowed us to compare our framework's performance with Simian against its native performance. For evaluation with the shuffling framework, a subset size of 50,000 files was chosen (58 subsets per round). Simian's fast execution allowed us to execute 30 rounds of the shuffling framework. Simian reports clones as clone classes, which was exploited when analyzing the results. We converted the clone classes into clone pairs to measure total recall.

Subset generation and round detection took approximately 8–12 and 4–10 h per round, respectively, on Westgrid hardware. Based on our later experiments, we estimate that subset generation and construction would take approximately 1.25 h, and detection approximately 1.25 h per round, on our local machine with a solid-state drive. The bottleneck on the Westgrid systems was the input/output performance. Specifically, copying many small files from the dataset into the subsets was much slower on a spinning hard disk drive.

7.1.2. Analysis. Because Simian's gold standard is extremely large (300 billion clone pairs), total recall was estimated using the heuristic, which is shown in Figure 5. After 30 rounds of the framework, 70% of the cloned fragments in Simian's gold standard were detected. According to the heuristic study (Section 5.4.2), total recall should be less than the heuristic, but with faster growth.

Table II. Summary of the IJaDataset clone detection experiments.

Tool	Hardware (GB)	Subset size (no. of files)	No. of sets	No. of rounds
Deckard	24	10K	289	10
NiCad	12	10K	289	20
Simian	12	50K	58	30

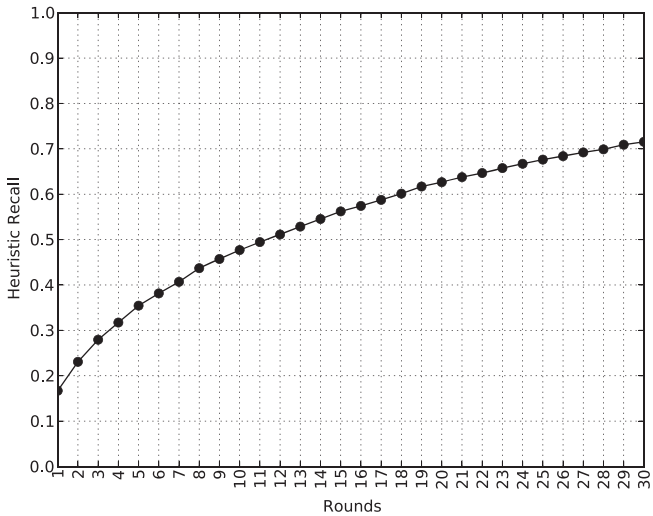


Figure 5. Simian heuristic (clone fragment) recall.

DOI: 10.1002/smr

Specifically, for Simian, the study showed that recovered recall quickly approached heuristic recall when total recall was within 70–90%. From Simian's heuristic recall trend for IJaDataset, we estimate it would reach 80% in approximately 10–20 additional rounds. We therefore conclude that Simian has achieved an acceptable recall for cloned fragments within 30 rounds. The heuristic recall suggests that within 10–20 additional rounds, a transitive clone recovery technique could achieve an equivalent total recall of clone pairs.

While the heuristic is a worthy approximation of total recall, it is still desirable to directly measure total recall, which necessitated a reduction in Simian's output. Investigation into the characteristics of Simian's gold standard found that 99.99% of Simian's clones came from clone classes greater than 100 fragments in size. Manual investigation into these clone classes revealed that Simian suffered from what we term the 'sliding effect'. It reported some extremely large clone classes containing the same fragment(s) repeated numerous times with small offsets in line numbers. These clone classes generate an extreme number of self and overlapping clones and represent a significant threat to Simian's precision. We therefore reduced Simian's output size by trimming clone classes over a certain maximum size. Its gold standard was likewise trimmed. The remaining clone classes were converted to clone pairs to measure total recall. This post-processing of the framework's output for Simian was performed solely to aid the evaluation of total recall on our hardware in a reasonable time frame and is not an expected post-processing step for users of the framework.

Figure 6 shows our framework's total recall with Simian for various maximum clone class sizes up to 100 fragments (limitation of our hardware). The legend of this figure specifies the maximum class size considered with the gold standard's size in parenthesis. Total recall was higher and increased faster for lower maximum clone class size. This suggests that the framework works best for specialized clone detection (i.e., focusing on detecting interesting/unique clones rather than all clones). This is due to larger classes requiring more rounds (on average) to be completely found as they contain more clone pairs that need to be shuffled together.

For the smaller class sizes a respectable total recall was achievable within 30 rounds (2: 52%, 5: 44%, 10: 40%). This total recall may be acceptable in cases where only a sample of the clones is required. For example, when building an inter-project clone corpus using many tools, 50% of a tools' native recall is likely sufficient for the corpus to benefit from the tool's unique detection characteristics. Consulting multiple tools may make up for individual tools' diminished recall. While this total recall is low, in each case it increases nearly linearly, with very little decay in slope. Additional rounds could bring these to an acceptable level. As can be seen, a 7–10% increase in total recall is gained per additional 10 rounds. A transitive recovery method could also help boost total recall achieved. In our preliminary studies, we found that the framework performed the worst with Simian. Therefore we expect the other tools to achieve a higher total recall than Simian.

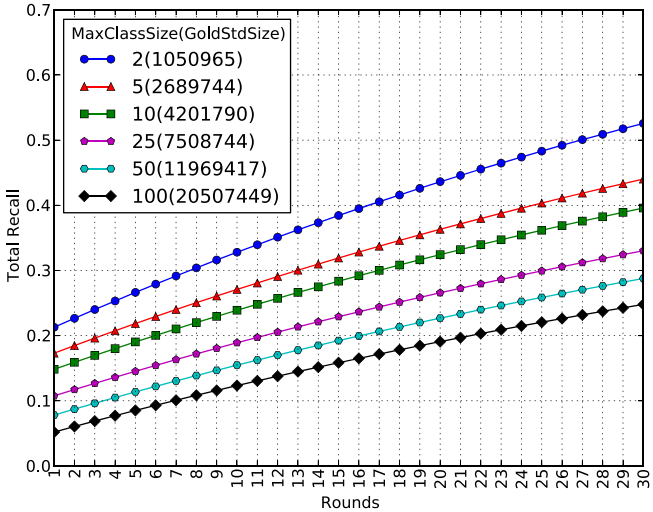


Figure 6. Simian total recall for maximum-class-size trimmed output.

DOI: 10.1002/smr

Figure 7 shows heuristic recall for the same trimmed output. As can be seen, the shuffling framework is finding the cloned fragments very fast, with 52–62% heuristic recall after only 30 rounds. Heuristic recall increases faster for larger maximum clone class size, meaning that the fragments in large clone classes are more easily found. This is expected as fragments in large clone classes have a higher chance of being shuffled into a partition with another fragment from the clone class. This suggests that a transitive recovery method may work especially well for the clones of large clone classes. This is particularly beneficial as it was for the clone pairs in larger classes that the framework had a slower increase in total recall (Figure 6).

7.2. NiCad

7.2.1. Setup. NiCad was included in this experiment for its ability to restrict clone detection to function clones. This is ~~a~~ beneficial for big data clone detection as line level clones may be too numerous to process. Function clones are fewer and may be more interesting as they occur at a higher level of software design. Function clone corpora built from big data inter-project datasets may be especially useful for mining new APIs.

Through experimentation, it was found that NiCad could consistently handle datasets of 10,000 files. It occasionally failed for larger input (e.g., 25K and 50K) because of hard-coded limits in the sizes of its internal data structures. These internal limits appear to be intentional and designed to prevent users from beginning executions that are likely to fail or never complete on standard workstations. The internal data structure sizes cannot be specified by the user without source code modification and recompilation. We left NiCad as is ~~for~~, our goal is to scale the tools without modifications.

Based on these observations, a subset size of 10,000 files was chosen for running the shuffling framework (289 subsets per round). As the framework achieved better total recall with NiCad than with Simian in the preliminary experiments and previous work [1], 20 rounds were deemed sufficient for demonstration of the framework. Subset generation and detection took 7–15 and 23–31 h per round, respectively, on shared computing resources. Based on our later experiments, we estimate that generating and building the subsets would require 1 h per round, and detection 17 h per round, on a solid-state drive.

7.2.2. Analysis. Creating a gold standard for NiCad was not possible, so we could not evaluate total recall. Internal data structure limits prevent NiCad from being executed on such a large input. Even if we modified these limits, NiCad would have required more RAM than we had available and likely months of execution time. We investigated using a deterministic partitioning technique (Section 2) to build NiCad's gold standard, but our estimates found that this would have required 2 months of

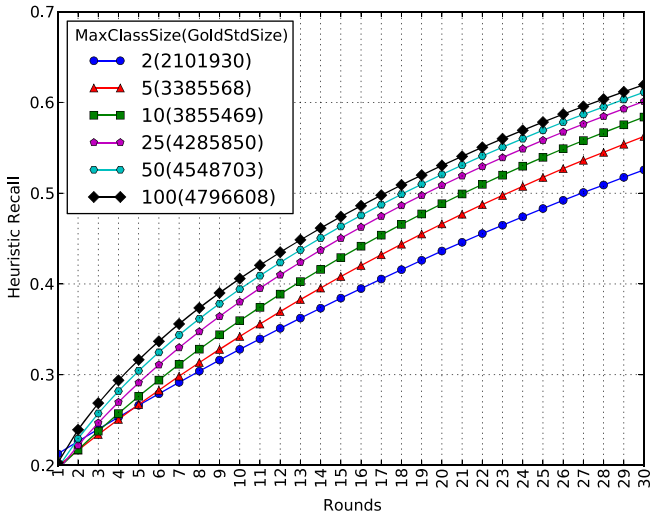


Figure 7. Simian heuristic recall for maximum-class-size trimmed output.

DOI: 10.1002/smr

execution time on our available hardware, even with four NiCad instances executing in parallel. Instead, we investigated the growth of the cumulative number of unique clones and cloned fragments found after each round of the framework. This information is plotted in Figure 8. In total, 5.66 million unique clone pairs containing 875,000 unique cloned code fragments were found.

The growth of unique detected clone pairs (Figure 8, diamond line) is linear across the 20 rounds. This tells us that the framework has not detected a large ratio of NiCad's native recall; the probability of undetected clones being shuffled together has remained constant over the rounds. Had a considerable ratio of NiCad's native recall been found, this probability should have also considerably decreased. Our preliminary study (Section 6) with small systems showed that the growth would appear logarithmic as the framework approaches a considerable ratio of the tool's native recall. Therefore, we require more rounds of the shuffling framework to achieve an acceptable ratio of NiCad's native recall.

In contrast, we do see logarithmic growth in the detection of unique cloned fragments (Figure 8, square line). Per round, the number of new cloned fragments found is decreasing noticeably. It is becoming less probable that a clone (in NiCad's native recall) that contains an undetected cloned fragment is randomly shuffled into a partition. This can only happen if a considerable ratio of NiCad's native cloned fragment recall is achieved per round. The growth has considerably declined after 20 rounds, suggesting that a considerable heuristic recall has been achieved.

These plots suggest that the framework is achieving a good heuristic recall with NiCad (the cloned fragments are being round quickly) but that the clone relationships between them (total recall) are still being detected. Applying a transitive clone recovery technique could recover some of the remaining clones without executing further rounds. As seen in the heuristic study (Section 5.4.2), clone recovery is very successful for NiCad. However, in that study, we applied transitivity naively to see the ideal results. To apply it in practice, an efficient and accurate method for checking the validity of transitivity for type 3 clones would need to be designed and implemented.

7.3. Deckard

7.3.1. Setup. Experimentation found that Deckard worked for our approach with a subset size of 50,000 files and could possibly work for larger subsets up to the entire dataset (untested). However, its execution time for large inputs was prohibitive (scaling limitation), so a subset size of 10,000 files was used to match NiCad (289 subsets per round). As Deckard has a lengthy execution time, the shuffling framework was executed ~~over~~ only 10 rounds. Detection was ran on Amazon EC2 and took approximately 5–7 days per round. While this execution time is very long, it is practical compared with Deckard's native execution time for the IJaDataset input.

7.3.2. Caveat. One disadvantage of Deckard is that it only supports up to Java 1.4 syntax. Its documentation specifies that it is able to skip unsupported syntax without error. In our experience, it found plenty of clones despite this limitation.

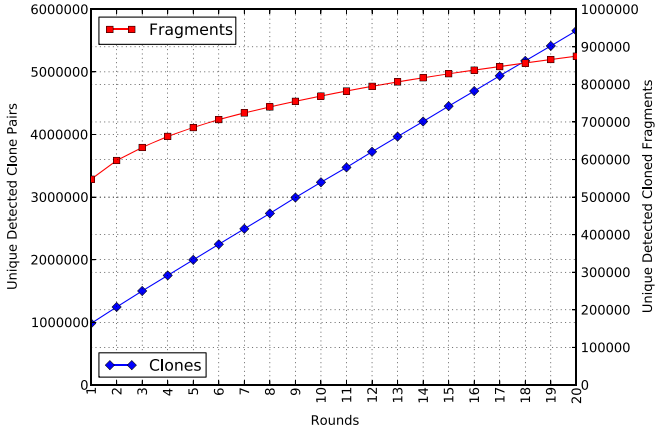


Figure 8. Growth of NiCad's found clones and cloned fragments.

DOI: 10.1002/smr

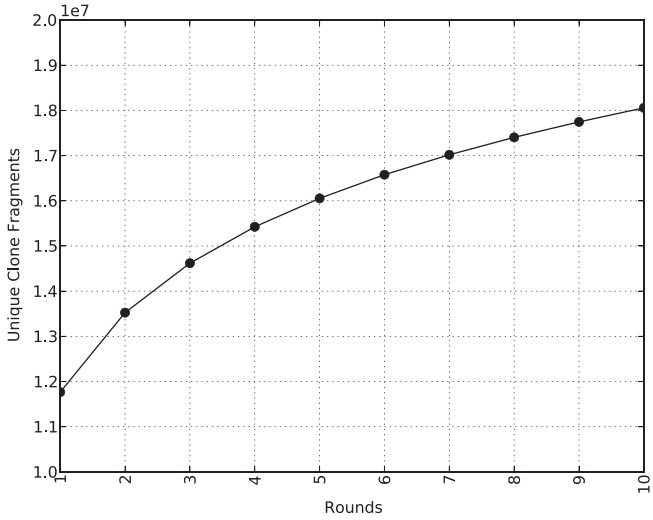


Figure 9. Growth of Deckard's detected cloned fragments.

7.3.3. Analysis. Creating a gold standard for Deckard was not possible because of the computation time required, so we could not investigate total recall. Instead, we investigated the number of unique clones and cloned fragments detected across the rounds as we did for NiCad.

Figure 9 shows the growth of the number of unique detected cloned fragments. As can be seen, the growth of detected cloned fragments follows roughly a logarithmic trend. The probability that the random partitioning shuffles a clone (in Deckard's recall) containing an undetected cloned fragment into the same partition decreases as heuristic recall increases. The considerable decrease in the number of new cloned fragments detected per round suggests that this probability is also considerably decreasing, and thus, a considerable heuristic recall has been found. Unfortunately, we could not measure the detected clone pairs across the rounds because of the size of Deckard's output. We can infer from NiCad's and Simian's results that it would likely be increasing linearly over these rounds.

In order to confirm our inference, we measured ~~found~~ clone pairs and fragments on a reduction of Deckard's output. We reduced the output ~~sized~~ by considering only the clone classes with a maximum size of 10 fragments (limitation of our hardware). The growth of detected clones and fragments for this reduced output is shown in Figure 10. As expected, we found very similar results to NiCad. The detected clones increase linearly, while the detected fragments show logarithmic

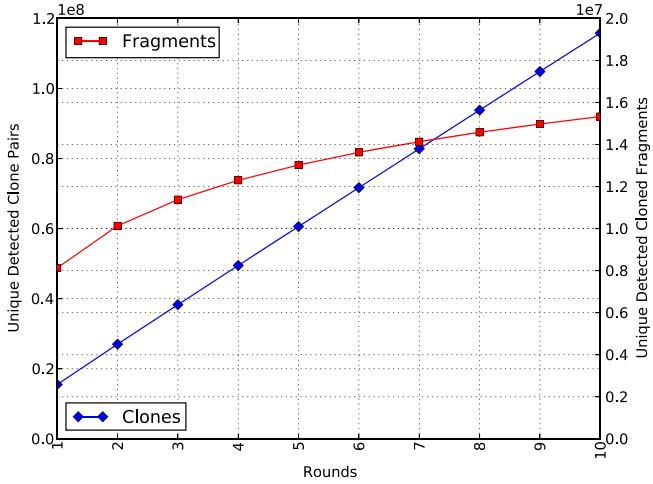


Figure 10. Deckard's clone and fragment detection for trimmed output.

DOI: 10.1002/smr

growth. This suggests that the cloned code fragments are being found before the clone pairs between them and that a transitive clone recovery method could be successful in detecting these clone pairs.

7.4. Other tools—SimCad, iClones, and CCFinderX

Our intention was to include SimCad, iClones, and CCFinderX in this experiment as they showed promise in the preliminary experiment. During evaluation of a sample from the dataset, these tools terminated without producing a clone report. SimCad and iClones reported encountering an invalid Unicode character. CCFinderX failed silently, but we believe this is because the dataset contains Java syntax of a newer specification than CCFinderX can parse.

These problems do not indicate special scalability issues with these tools or with our framework. However, the framework cannot make progress with tools that abandon detection when parsing errors are found, instead of trimming the offending file(s). Because we are executing these tools for partitions of the IJaDataset near the limits of their scalability, there is a high chance that they encounter at least one parsing error. The tools fail upon the first parsing error detected, so it is not practical to compile a list of offending files in order to trim the dataset.

Communication with the iClones developers revealed that this problem was fixed in a development branch, so we included iClones in our experiments in improving the shuffling framework (Section 9). SimCad was also corrected upon communication with the developer, but not in time to be used in this publication. We plan to revisit SimCad in future work. CCFinderX is no longer under active development, so we do not anticipate improvements in its parsing or error handling.

7.5. Summary

From these experiments, we found that the clones found by the framework increased nearly linearly, with a slight decay in slope, across the rounds. This shows that additional rounds would continue to see a healthy increase in found cloned pairs, and thus an increased total recall. For Simian and considering only clone pairs originating from smaller clone classes (2–100 fragments), 25–52% total recall was achieved over 30 rounds, with a (decaying) continued increase of 7–10% per 10 rounds (Figure 6). Further rounds could bring total recall to an acceptable value.

However, the framework was able to find the clone fragments much faster. For each tool, found cloned fragments experienced logarithmic growth across the rounds. The decay in detection rate indicates that the probability of a clone containing an undetected clone fragment randomly shuffled into a partition is decreasing noticeably. This indicates that the number of remaining undetected clone fragments is decreasing considerably. With Simian, 70% of the cloned fragments were found within 30 rounds (Figure 5).

These findings suggest that our framework finds most of the cloned fragments in few rounds but may require a large number of rounds to find all of the clone relationships between them. This suggests that a transitive-based clone recovery process could improve total recall achieved. This is supported by our heuristic study (Section 5.4.2), which showed that a strong heuristic (clone fragment) recall can be translated into a strong total (clone) recall by transitive recovery. Implementing an efficient and precise recovery process is therefore a priority for our future work.

From our experiment, we conclude that the shuffling framework is successful in scaling classical clone detection tools to ~~ultra-large~~ datasets (RQ#3), but many rounds may be needed to achieve a high total recall. The framework is best suited for applications that accept partial clone detection tool recall as sufficient. For example, when building a comprehensive inter-project clone corpus (e.g., for IJaDataset) using a variety of both classical and scalable detection tools, 60–80% of a classical tool's native recall is likely sufficient to ensure the clone corpus benefits from its diverse strengths and detection characteristics.

The framework is very suitable for applications that only require knowledge of the cloned fragments within ~~an~~ ~~ultra-large~~ dataset, and not the pairs. Given that we encountered scalability limits (memory and time) in processing the clone pairs found by this experiment, it is likely that studies on inter-project clone corpora of similar scale may need to be carried out on cloned fragments. Analyzing the clone pairs presents an additional big data challenge.

DOI: 10.1002/smr

8. SHUFFLING FRAMEWORK PERFORMANCE ANALYSIS

As seen in the IJaDataset experiment, the shuffling framework is able to scale classical tools to large datasets. It is able to tackle various scalability issues, including memory requirements, computation complexity, computation time, and internal tool limitations. However, we observed some inefficiencies in the original algorithm.

In the IJaDataset experiments with the core shuffling framework, a suitable clone fragment recall was obtained (e.g., Figure 5). However, clone pair recall was much lower (Figure 6). To obtain a higher clone pair recall, many more rounds would need to be executed, which would require considerable computation time. Alternatively, post-processing could be used to recover some of the missed clone relationships between the detected cloned fragments. Previously, we showed that clone transitivity is effective at clone recovery (Section 5.4.2). However, transitivity is only certain for type 1 and ~~2~~ 2 clones. Type 3 clones recovered by transitivity would need to be verified before accepted. Our experiences with the IJaDataset experiment indicate that post-processing may be computation and memory intensive. We had considerable difficulties processing the detection results for statistical reporting. A novel and efficient approach and considerable computer resources are likely required to apply transitive clone recovery with both high recall and precision.

We decided that the best way to improve the performance of the shuffling framework was to improve the shuffling algorithm itself (RQ#4). Our goal was to decrease the number of subsets of the dataset a tool had to be executed for to obtain an acceptable total recall. Looking at the tools' clone pair detection performance for IJaDataset (Figures 6, 8, and 10), we notice a common trend. A large number of clones are detected in the first round, followed by a lesser but steady increase in subsequent rounds. We studied this behavior and found that the large increase in the first round is due to the successful detection of all the intra-file clones in the tool's gold standard. This occurs because the first round exposes the clone detector to every file in the dataset. The remaining rounds advance the detection of the inter-file clone pairs in a tool's gold standard. Fewer subsets would be required if the shuffling algorithm focused on the detection of the inter-file clones in rounds 2 through n.

We identified two major characteristics of the core shuffling framework that slow the rate of inter-file clone detection. First, the shuffling framework has no sense of history. There is nothing to stop it from repeatedly shuffling the same files into the same subsets. The clone detector will find any inter-file clones between a pair of files (that it is able to detect) the first time they are shuffled together. Repeated shuffling of previously seen file pairs does not advance inter-file clone detection but uses computation time. An improved shuffling framework should discourage repeated shuffling of the same file pairs.

Secondly, the shuffling framework does not consider the contents of the files it shuffles together. Likely only a small ratio of the dataset contains inter-file clones. It is wasteful to shuffle dissimilar files together. The framework would require fewer subsets if it preferred to shuffle together files that are similar enough to likely contain a clone as judged by the classical tool. In the following section, we explore incremental improvements to the shuffling algorithm that address these two limiting characteristics.

9. IMPROVING THE SHUFFLING FRAMEWORK

We identified in the previous section that the shuffling framework's performance suffers from the shuffling together of the same files repeatedly and the shuffling together of dissimilar files unlikely to contain clones. Addressing these problems is non-trivial as optimal solutions are not practical for big data due to the high complexities of the required algorithms. Even sub-optimal solutions can quickly increase the complexity and execution time of the shuffling algorithm. Our goal is to trade detection execution time (i.e., fewer subsets) for shuffling execution time (i.e., more valuable subsets). For this to provide a performance gain, the shuffling algorithm needs to maintain a lower complexity and execution time than that of the clone detection tools. That is, the cost of building subsets that provide a larger increase in total recall must be less than the cost of simply executing the tool for more subsets.

DOI: 10.1002/smr

We investigated three new shuffling algorithms that incrementally address the issues of the core algorithm. These include the unseen-pair, the unseen-similar-pair, and inverted-index shuffling algorithms. We now term our original algorithm the blind partitioning shuffling algorithm, because it builds its subsets by blind random partitioning of the dataset.

The three new algorithms use two rounds of shuffling. In the first round, the framework completely partitions the dataset into disjoint subsets, and the tool is executed for each of these subsets. This is the same as a round generated by the original core algorithm (Section 4). This first round exposes the tool to every file in the dataset, which ensures that the framework does not miss any of the intra-file clones the tool is able to detect. In the second round, the framework pursues the inter-file clones the tool is able to detect. Because the intra-file clones should have been detected in the first round, it is no longer important to expose the tool to every file in the dataset in round 2. Therefore, the new algorithms drop the partitioning strategy in round 2. Instead, they execute the tool for a series of subsets that prioritize the exposure of the tool to new inter-file clone detection experiences. These subsets may not completely partition the dataset, they may overlap, and round 2 may contain any number of subsets. The subsets of round 2 are the same size as those of round 1, and non-determinism is still exploited in their selection. The difference between the algorithms is in how they choose these subsets. Their goal is to require fewer subsets (fewer executions of the tool) than the blind partitioning algorithm to achieve some target total recall.

We evaluated these algorithms for three clone detection tools: NiCad, iClones, and Simian. NiCad and Simian were used in our ~~previous experiments~~. We now include iClones, which had been fixed after our IJaDataset experiment (Section 7) had been conducted. The shuffling framework performed very well with iClones in the preliminary experiments (Section 6), so it was ideal to include it as a subject tool in this experiment. We decided to skip Deckard because while the shuffling framework was able to improve its scalability, it still required long execution times.

We evaluated the algorithms using random samples of the IJaDataset. We choose sample sizes large enough to be more representative of ~~a big data~~ than the systems used in the preliminary experiments (Section 6), while still small enough that we could obtain each tool's gold standard for measuring total recall. For NiCad and Simian, we used a dataset of 50,000 files randomly selected from the IJaDataset. For iClones ,we randomly selected 10,000 IJaDataset files. For the 50,000-file sample, a subset size of 250 files was used, and 50-file subsets were used for the 10,000-file dataset. This is the same ratio between subset size and dataset size as used in the IJaDataset experiment (Section 7) for NiCad and Deckard. This way, the evaluation of the algorithms approximates their usage with IJaDataset, or another a big data dataset.

Performance was measured as total recall, the ratio of the clone pairs that the shuffling algorithm was able to find in a tool's gold standard (Section 5, Eqn. 1). In order to save time, we simulated the execution of the tools. We assumed that for a particular subset, the tool would output the clones from its gold standard that are within or between files in the subset. For these deterministic code clone detection algorithms, this is a reasonable assumption.

By simulating the clone detection, we are able to measure the framework's performance more accurately. Ideally, a clone detection tool reports the same clones between a pair of files regardless of the number and particular files also included in the input. In practice, the clone detector may not report clones consistently. It may report the same clones but with slightly different start/end lines, or it may miss some clones or find additional clones. Its precision (the number of false positives reported) may also vary. This may be caused by bugs in the clone detector. By simulating the detection using the clone detector's gold standard, we avoid these issues.

The total recall performance of these four algorithms are shown in Figures 11–13. Their subset generation time is shown in Figures 14 and 15. The generation time includes only the time required to generate the list of files to be included in each subset. It does not include the time needed to copy the files into a temporary directory and execute the tool.

In the following subsections, the algorithms are outlined, and their performance discussed. The algorithms are presented in the order in which they were created so as to emphasize our design process and decisions.

DOI: 10.1002/smr

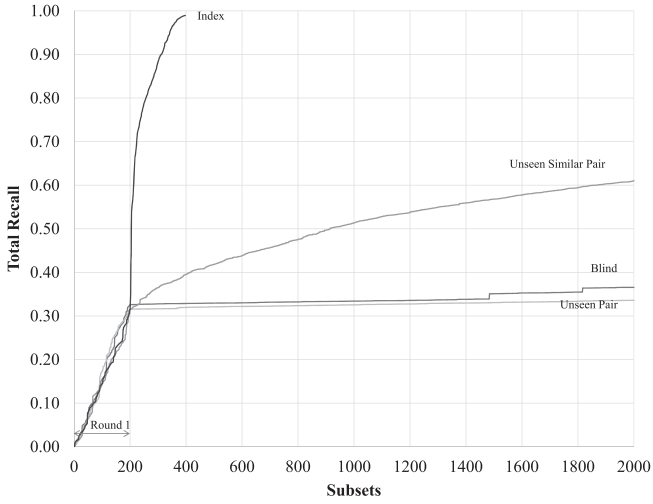


Figure 11. Shuffling algorithm performance comparison: NiCad, 50,000-file dataset, 250-file subsets.

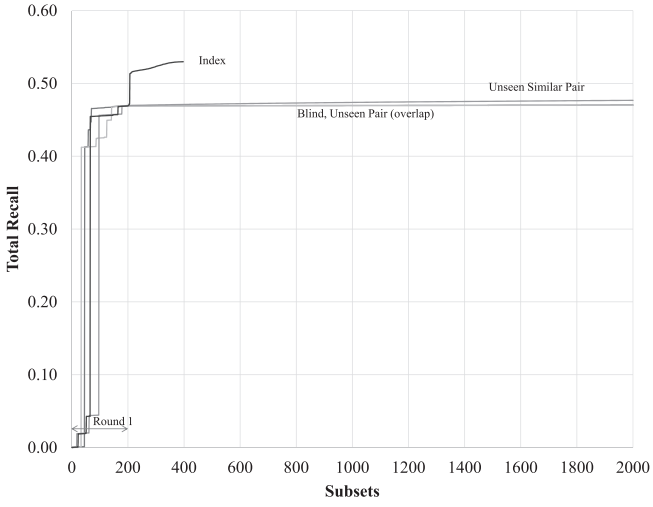


Figure 12. Shuffling algorithm performance comparison: Simian, 50,000-file dataset, 250-file subsets.

9.1. Blind partitioning shuffling algorithm

The blind partitioning shuffling algorithm is the original shuffling algorithm as presented in Section 4. It is the cheapest shuffling algorithm in terms of subset generation processing time and complexity. Its performance is the baseline against which the other algorithms are compared. For both the 10,000-file and 50,000-file datasets, the blind shuffling algorithm partitioned the dataset into 200 subsets per round for 10 rounds. Remember that for each round, this algorithm partitions the dataset into mutually exclusive subsets that together span the entire dataset. The result of its application for NiCad, Simian, and iClones can be seen in Figures 11–13, respectively. Because the new algorithms do not use the same number of rounds, the total recall is plotted after each subset. The tick marks of the *x*-axis correspond to the rounds of the blind shuffling algorithm. The first round (in which all algorithms use the blind partitioning strategy) is labeled.

This algorithm achieves a large total recall increase for all tools within the first round. Because the first round exposes the clone detector to every file in the dataset, all the intra-file clones in the gold standard should be detected in this round. In the remaining rounds, we see linear growth in total recall as the inter-file clones in the gold standard are located by chance owing to blind random file shuffling. Linear growth was expected as the algorithm applies the same random dataset partitioning in each round. This linear growth should experience a decay in its slope as more clones are

DOI: 10.1002/smr

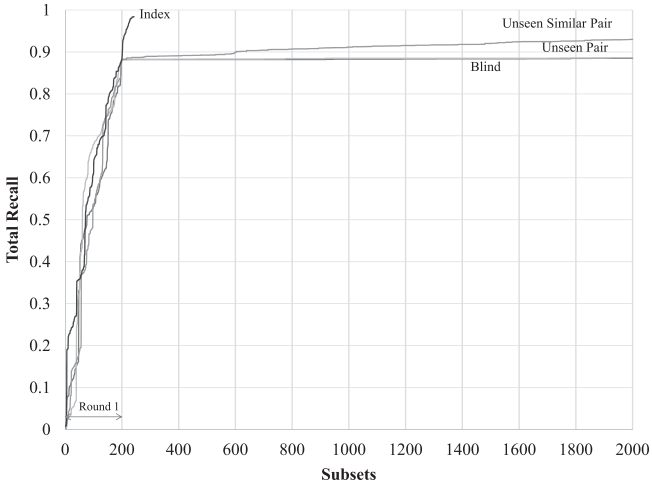


Figure 13. Shuffling algorithm performance comparison: iClones, 10,000-file dataset, 50-file subsets.

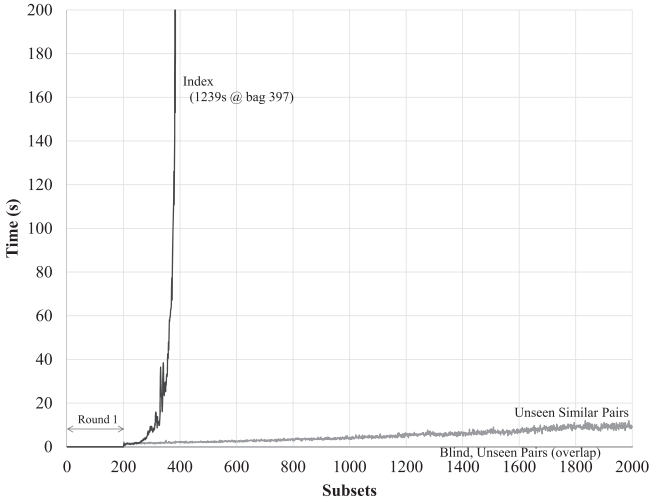


Figure 14. Shuffling algorithm computational comparison: subset generation time (ms), 50,000-file dataset (NiCad/Simian).

detected, and it becomes increasingly less likely that the random partitioning will shuffle the files containing the undetected clones into the same subset.

The increase in total recall due to inter-file clone detection in rounds 2 through 10 is much smaller than the increase due to inter-file and intra-file clone detection in round 1. Because the growth is linear across rounds 2 through 10, we expect that roughly the same number of inter-file clones was detected in round 1. Therefore, the larger increase in total recall in round 1 must be dominated by the detection of the intra-file clones.

9.2. Unseen-pair shuffling algorithm

The first problem we identified with the blind shuffling algorithm is that it does not discourage the repeated shuffling together of the same files. Our efficient solution is to fill the subsets with randomly selected pairs of files from the dataset that have not been shuffled into the same subset previously. This solution guarantees that each subset includes $n/2$ new inter-clone detection experiences (i.e., unseen file pairs), where n is the number of files in the subset. This is the same number guaranteed by a deterministic approach (Section 2). However, in this non-deterministic case,

DOI: 10.1002/smr

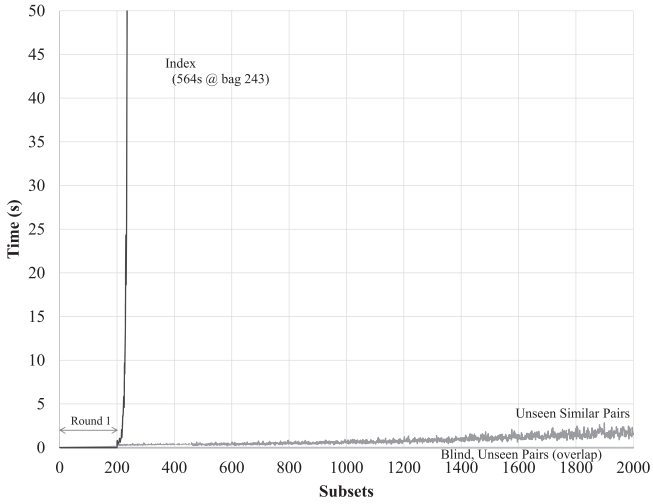


Figure 15. Shuffling algorithm computational comparison: subset generation time (ms), 10,000-file dataset (iClones).

it is likely that the files in a subset form many additional unseen pairs other than those specifically chosen. The worst case of only $n/2$ unseen pairs in a subset should only occur once most of the file pairs in the dataset have been seen in earlier subsets.

The first round of this algorithm uses blind shuffling to partition the dataset into subsets. This is needed as the unseen-pair strategy does not guarantee nor encourage the detection of all intra-file clones in the dataset. By the end of the first round, every file has been seen by the clone detector. This first round does not detract from the unseen-pair strategy as all subsets of the first round are made up completely of unseen pairs.

For round 2, the algorithm fills a user-specified number of subsets with unseen file pairs. Specifically, for each subset, the algorithm performs the following steps:

1. Two files are randomly chosen from the dataset.
2. The algorithm efficiently checks if the pair has been seen in a previous subset.
3. If not previously seen, the files are added to the current subset.
4. If the subset is not full, the algorithm repeats from step 1. Otherwise, the subset is complete.
5. The specification (file list) of the subset is saved.
6. The algorithm repeats from step 1 for some number of subsets.

This subset-filling technique is efficient until most of the pairs have been seen, at which time, the algorithm may cycle extensively until an unseen pair is found. For this reason, the algorithm is parameterized with a stopping condition: the number of times to cycle before giving up on being able to fill the subset. However, for a big data dataset, it is unlikely that this will occur for a practical number of subsets.

This algorithm needs to be able to efficiently check if two files have been seen together in a previous subset, step (2). This is accomplished by assigning each file in the dataset an id and tracking the contents of each subset using a bit vector. Vector[i](id) is 1 if the file with the specified id is in the ith subset. Shuffling a big data dataset like IJaDataset, which contains approximately three million files, requires only 0.35 MB of memory per subset. The algorithm can determine if a file pair is unseen, in the worst case, in $O(b)$ time, where b is the number of previously generated subsets. Examining a bit vector is very fast, so this linear search is acceptable. There are ways to structure the seen-pair data to make $O(1)$ possible, but it requires too much memory for the data structure to fit in RAM. We use this linear approach to avoid disk access times.

The unseen-pair shuffling algorithm's total recall for the three tools is shown in Figures 11–13, and its subset generation time is shown in Figures 14 and 15. Remember that the first 200 subsets (round 1) are built using the blind partitioning strategy to ensure intra-file clone detection. This algorithm performed a little worse than the blind shuffling algorithm for the NiCad experiment, equally for the

DOI: 10.1002/smr

Simian experiment, and a little better for the iClones experiment. Its generation time is also essentially equivalent to that of blind shuffling.

Overall, the unseen-pair shuffling algorithm does not perform any better or worse than blind shuffling. The reason for this is quite simple: the number of pairs in these datasets is so numerous that the blind shuffling technique is not shuffling the same files together repeatedly as frequently as we feared. However, the unseen-pair strategy would be required if the number of potential file pairs was reduced. We decided to extend this algorithm to consider the second problem with blind shuffling: we should only shuffle together files that are similar. As this will likely reduce the number of potential file pairs considerably, we maintain the unseen-pair selection strategy in the next algorithms.

9.3. Unseen-similar-pair shuffling algorithm

The unseen-similar-pair algorithm extends the unseen-pair algorithm to also address the second problem with the blind shuffling algorithm: it does not consider the contents of the files it shuffles together. Similar files are more likely to contain clones, so total recall obtained per subset could be achieved by prioritizing the shuffling together of similar files. Specifically, our goal was to parameterize the unseen-pair algorithm with a file-similarity heuristic. Under this scheme, unseen file pairs are added to a subset only if the heuristic decides the files are similar enough to possibly contain a clone as judged by the classical clone detection tool.

The challenge was in designing a suitable heuristic that can make a smart decision without significantly adding to the shuffling algorithm's complexity and execution time. Most clone detectors report a clone between two files if they share a sequence of similar source code lines of some minimum size. Therefore, if two files contain this minimum number of similar lines, then it is possible the tool will find a clone between them. However, searching for similar line sequences between two files has polynomial complexity, which is also the complexity of most clone detectors. We needed a linear heuristic to ensure the improved shuffling algorithm was a performance gain.

Our heuristic accepts a file pair if the files share a minimum number of similar source lines. To do this~~,~~ in linear time, we dropped the requirement that the similar lines need to be sequential. The disadvantage of this speedup is that the heuristic will accept file pairs with similar lines too sparsely distributed to be a clone. As source line (string) comparison is costly, we pre-computed hash code (integer) values for each source line in the dataset. The number of shared lines between two files can then be calculated by measuring the size of the intersection of the hash codes they contain.

To improve the heuristic's clone presence detection and accuracy, we normalized and filtered the source code during the hash coding process. First, the source code was pretty-printed to normalize formatting. Identifiers were normalized (blind renaming) such that two source lines that differ only by identifier names will hash to the same value. Inconsequential, but common, source lines (e.g., '}') were removed to reduce the heuristic's false-positive rate. Comments were also removed before hashing as most clone definitions and detection tools ignore them.

The heuristic is parameterized with the minimum clone size (in identical lines) of the target tool's configuration. For example, if the tool is set to report clones 10 lines or larger with a minimum of 70% similar lines, then the heuristic should be configured for seven identical lines. Specifically, this is the minimum number of identical lines in a type 3 clone as judged by this tool. The heuristic will then accept file pairs that could contain a clone as judged by the tool and reject those that do not contain sufficient similar lines for the tool to report a clone.

Mismatch between how the framework and the tool count source lines may cause some file pairs containing a clone detectable by the tool to be rejected. Mismatch may occur because of differences in how the framework and the tool normalize and filter input source code. Also, when tools measure minimum clone size by tokens, ~~then~~ minimum lines need to be estimated. Rejected file pairs that contain a clone as judged by the tool will lead to additional false negatives. This should only occur for small clones that are near the minimum clone size and whose files contain no other similar lines. The latter because the heuristic does not consider the position of the lines when measuring the number of lines shared between two files. This is acceptable as smaller clones are more likely to be spurious or uninteresting. Compensating for mismatch by setting the minimum clone size lower than that of the tool is a bad idea as it will cause the heuristic to accept more file pairs that do not

DOI: 10.1002/smr

contain detectable clones, which lowers the effectiveness of the heuristic. With the heuristic, we trade some of the tool's recall for smaller clones in exchange for fewer subsets to meet a target portion of the tool's native recall (total recall).

For the evaluation of this algorithm, we parameterized the heuristic with a minimum similar-line threshold of five lines. By default, Simian detects type 1 and 2 clones with a minimum of six lines, iClones a minimum of 100 tokens (~5–10 lines) with gaps, and NiCad a minimum of 10 lines (30% of which may be gap lines). The five-line threshold should be conservative enough to not skip too many file pairs that contain clones as judged by these tools. The algorithm's total recall performance is shown in Figures 11–13, and its subset generation time is shown in Figures 14 and 15. Again, its first 200 subsets are its first round of blind shuffling to allow full intra-file clone detection.

For our 50,000-file dataset, subset generation time begins at approximately 1.5 s and increases linearly as more subsets are generated. The generation time increases as the number of remaining unseen and similar file pairs in the dataset decreases. It takes longer to randomly locate an eligible file pair as they become more rare. This is not a defect as the rarer the eligible pairs become, the higher the total recall the shuffling algorithm has obtained. The number of subsets to generate is therefore a balance between the total recall goal and available subset generation time.

For all three tools, the unseen-similar-pair algorithm achieves a higher total recall than both the blind partitioning and unseen-pair shuffling algorithm within the same number of subsets. This increase is most pronounced with NiCad and iClones, while only a small increase is achieved with Simian. The algorithm's total recall for Simian may be higher if a more conservative similar-line threshold were used.

Smaller gains with Simian might indicate the tool has worse precision. This shuffling algorithm only encourages the shuffling together of file pairs that may contain a true-positive clone. The measurement of total recall does not consider if the clones in the tool's gold standard are true or false positives. Precision deficiencies of the tool would manifest in this evaluation as poorer total recall measurement. However, we do not have sufficient information about Simian's precision to conclude this.

9.4. Inverted-index algorithm

The unseen-similar-pair algorithm successfully increases the performance of the shuffling framework. However, we believed that considerably better performance could be achieved if the heuristic was sensitive to the locations of the shared source code between the files. The heuristic needed to be able to detect if two files had similar lines that were also closely located (e.g., subsequent). We were able to achieve this without increasing the complexity of the heuristic by computing *n*-grams across the hashed source lines of the dataset that were used with the previous algorithm.

The *n*-grams were calculated by summing each *n* subsequent hashed source lines using a sliding window. For example, a file with the hashed source lines *A*, *B*, *C*, *D*, and *E* has a 3-gram representation of $(A+B+C)$, $(B+C+D)$, and $(C+D+E)$. The heuristic then approves a file pair if they have a minimum number of similar *n*-grams between them. For our evaluation of this algorithm, we used a 3-gram representation. We pre-computed this in linear time in a single pass across the hashed version of the dataset.

The heuristic was parameterized to accept file pairs with at least three shared 3-grams. In effect, the heuristic considers two files to contain a clone if they share at least three incidents of three similar and subsequent original source lines, which may overlap. File pairs approved by the heuristic therefore have at minimum between five (totally subsequent) and nine (three incidents of three subsequent) similar source lines.

During our initial investigation of this algorithm, we found it had a very lengthy subset generation time. It was spending a large amount of time randomly selecting file pairs from the dataset that did not satisfy the heuristic. We minimized this problem by selecting the file pairs from an inverted file index built for the *n*-grams. The index maps each *n*-gram value to the files that contain at least one incidence of that *n*-gram. By randomly selecting file pairs from the index, we are guaranteed they share at least one *n*-gram. This considerably reduces the selection space and allows the subsets to be built faster. The inverted index was represented by a hash map and built in linear time by a single pass across the *n*-grams.

The inverted index can still be too large of a search space. Some *n*-grams appear very frequently within the dataset. To counteract this, the index is trimmed of the *n*-grams that appear in over a maximum number

DOI: 10.1002/smr

of files. Less common *n*-grams are more likely to denote a clone rather than common structural/stylistic code (e.g., a series of declaration statements at the beginning of a function). For our evaluation, we set the *n*-gram appearance threshold for our inverted index to 1000 files.

The evaluation of this algorithm with NiCad, iClones, and Simian for 3-grams, a minimum of three shared 3-grams between file pairs, and an index *n*-gram appearance threshold of 1000 files are shown in Figures 11–13. The subset generation times are shown in Figures 14 and 15.

Subset generation time increased exponentially as more subsets were created. We stopped the generation after 397 subsets (the first 200 of which were the round 1 blind shuffling subsets) because the subset generation time had increased by two orders of magnitude. The high generation cost means that file pairs that both satisfy the *n*-gram similarity heuristic and remain unseen have become rare in the search space (inverted index). It is taking a long time for the random selection process to find a suitable pair. The fact that the algorithm is reaching a high generation cost so quickly means that it is reaching its maximum total recall potential in fewer subsets than the other algorithms.

Not only does this algorithm exhaust its search space in fewer subsets, these subsets also provide larger increases in total recall. With NiCad and iClones, this algorithm achieved a very high total recall using far fewer subsets than the other algorithms. With these tools and given a sufficient number of subsets, nearly 100% of the tool's native recall was achieved. Considerable gains were also seen with Simian compared with the other shuffling algorithms, but end total recall was much lower. The heuristic settings may not have been conservative enough for the types of clones Simian detects. Total recall would also be low if Simian has low precision. This shuffling algorithm avoids shuffling together files that are not similar as judged by the file-similarity heuristic. An inter-file false positive in Simian's gold standard may never be shuffled together. However, we cannot conclude this as we do not know Simian's precision performance for large inputs.

Because subset generation time increases so rapidly, we decided to investigate how quickly it increased with respect to total recall achieved. We plot this for NiCad and Simian (50,000-file dataset) in Figure 16 and for iClones (10,000-file dataset) in Figure 17. With NiCad, subset generation time had only increased by a single order of magnitude (1–10 s) by the time a 90% total recall was achieved. This is up from 32%, the total recall after blind partitioning in round 1. To reach a nearly 100% total recall, another order of magnitude increase in subset generation time was required. The exponentially increasing cost of subset generation only becomes severe after most of the total recall has been achieved, which is very acceptable.

We see a similar trend for iClones. Total recall increases from 88% to 98% within the first order of magnitude increase in subset generation cost. We then see very little gains in total recall for the next order of magnitude increase. The primary difference from NiCad is that the framework had a high total recall with iClones after the first round (blind partitioning). This was because we had to use a

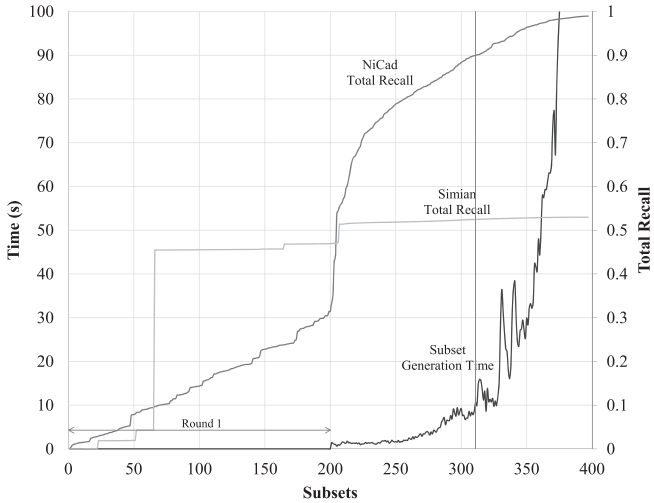


Figure 16. Inverted-index algorithm—subset generation time versus total recall (NiCad/Simian, 50,000-file dataset).

DOI: 10.1002/smr

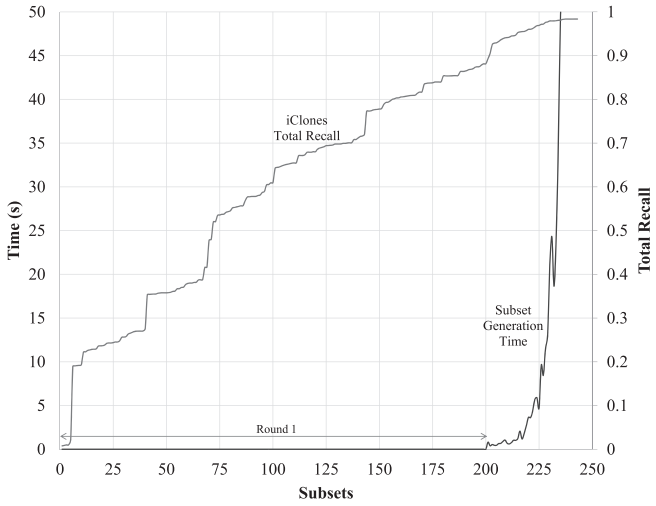


Figure 17. Inverted-index algorithm—subset generation time versus total recall (iClones, 10,000-file dataset).

dataset five times smaller for this experiment with iClones, owing to iClones memory requirements. A smaller dataset will have a larger intra-file-to-inter-file clone ratio. The average incidence of intra-file clones for samples of the IJaDataset will remain constant no matter the sample size. However, for larger samples, the average incidence of inter-file clones will be higher because the number of file pairs potentially containing clones increases polynomially with sample size.

9.5. *Choosing an algorithm*

Of the four algorithms proposed, the inverted-index shuffling algorithm achieves the highest total recall in the fewest subsets. Its subset generation time is much longer because of its smaller search space, but this is an advantage because it is converging to a high total recall quickly. This equates to far fewer executions of the clone detection tools, which have higher complexities and longer execution times than subset generation. Using our simulated experiments using smaller datasets, we have demonstrated success for RQ#4: by observing the performance of the shuffling framework and incrementally responding to our observations, we were able to significantly improve its performance.

10. THE IMPROVED SHUFFLING FRAMEWORK

In this section, we summarize the improved shuffling framework as developed in Section 9. The improved shuffling framework executes a classic tool for two rounds of subsets of some big data ~~input~~ source code dataset. The first round uses the blind partitioning shuffling algorithm, while the second round uses the inverted-index shuffling algorithm. Consider the tool's hypothetical gold standard for the big data dataset. Round 1 will enable the tool to detect all of the intra-file clones in the gold standard (in addition to some inter-file). Round 2 aims to enable the tool to detect a large ratio of the remaining inter-file clones in the gold standard in as few subsets as possible. The steps of the improved framework are summarized in Figure 18.

We demonstrated in Section 9 that this method can achieve up to 98% of a classical tool's native recall given a sufficient number of subsets. However, subset generation became very expensive when 90% of the tool's native recall was achieved. The improved shuffling framework must generate the subsets serially as it needs to track which pairs of files have been seen in a previous subset, necessitating a definite subset order. However, the clone detection tool can be executed as soon as the first subset has been generated.

The improved framework guarantees that each subset contains at least $m/2$ unseen file pairs, where m is the size of the subsets. However, until the majority of the search space has been investigated, the subsets should contain far more unseen pairs. The similarity heuristic ensures that the guaranteed

DOI: 10.1002/smr

1. Preparation
   (a) The maximum input size measured in source files, m, a clone detection tool can reliably handle on standard hardware is measured.
   (b) A hashed version of the dataset is generated. Each source line is replaced by an integer hashcode. The dataset is normalized (pretty-printed, identifier renaming) and filtered (of common structural lines) before hashing.
   (c) The hashed version is replaced by sliding n-grams, for some value n.
   (d) An inverted index is built, which maps each n-gram value to the files that contain at least one incidence of that n-gram.
   
   The t most popular n-grams are removed from the inverted index, in order to reduce the search space.
2. Subset Generation: Round 1 - Blind Partitioning
   (a) The source files of the dataset are randomly partitioned into non-overlapping subsets of size m.
   (b) The specification (file list) of each subset is stored.
3. Subset Generation: Round 2 - Inverted Index
   
   A subset of size m is constructed by randomly selecting pairs of files from the inverted index that share at least one n-gram. A pair is added to the subset if (1) the files have not previously been in the same subset, and (2) the files share at least s n-grams.
   (b) Once the subset is full, its specification (file list) is stored.
   (c) Subsets are generated until either (1) some user-specified maximum has been reached, (2) some maximum number of randomly selected file pairs have been rejected in a row, or (3) the user interrupts the process.
4. Clone Detection
   (a) A subset's specification is retrieved and the subset is constructed.
   (b) The tool is executed for the subset.
   
   The tool's clone detection report is merged into a clone repository that efficiently removes duplicates (e.g., hash set, indexed database table).
   (d) Clone detection on the subsets may be done serially, in parallel or distributed. Clone detection can begin as soon as the first subset has been generated. A subset can be analyzed as soon as its specification has been completed.

Figure 18. Improved shuffling framework procedure (summary).

unseen file pairs also contain enough similarity to possibly contain a clone. This is performed by selecting an *n*-gram size, *n*, and minimum shared *n*-gram heuristic, *s*, with respect to the tool's minimum clone size in identical source lines. If two files contain exactly *s* *n*-grams in common, then they share between $n+s$ (sequential) and $n*s$ (*s* occurrences of *n* sequential) source lines. So *n* and *s* should be picked with respect to the tool's minimum clone size. We had good results using 3-grams. A trim threshold for the inverted-index, *t*, must also be selected. We had good results trimming the index of any *n*-gram that appears in over 1000 files.

10.1. Comparison with deterministic method

The improved shuffling framework is a non-deterministic method for scaling classical tools. The most similar previous work to our framework was the deterministic method described in Section 2. The deterministic method scales tools by doing the following: (1) partitioning the dataset into partitions half the size of the tool's maximum input and (2) executing the tool for each unique pair of partitions. The deterministic method achieves 100% of a tool's native recall after $x\frac{(x-1)}{2}$ subsets. x is the number of partitions and $x = \frac{r}{(0.5\mathrm{m})}$, where r is the size of the dataset and m is the tool's maximum input size.

Consider if we split the deterministic method's subsets into two rounds, as we did with our non-deterministic shuffling framework. Both methods have $\frac{r}{m}$ (or $\frac{x}{2}$) subsets in their first round, and each of these subsets has $m\frac{(m-1)}{2}$, unseen file pairs. In the second round, the deterministic method has exactly $\frac{m}{2}$ unseen file pairs per subset. Each of the deterministic method's partitions has been seen in round 1, so the only unseen pairs are those between the joined partitions. Our shuffling framework guarantees that its subsets in round 2 contain *at least* $\frac{m}{2}$ unseen file pairs but should contain many more until the majority of the search space has been explored. The shuffling framework's similarity heuristic means that its subsets will contain more clones on average than the deterministic method.

DOI: 10.1002/smr

Our shuffling framework is faster (i.e., using fewer subsets) than the deterministic method. However, the deterministic method is better for some use cases. As seen in Section 9, the improved shuffling framework hits a point of diminishing returns before 100% total recall is achieved. The cost of generating the subsets becomes too costly. So the shuffling framework is appropriate for cases where some sacrifice in a tool's native recall is permissible. The deterministic method is needed when 100% native recall is required. Note that no clone detector has perfect recall, so 100% native recall does not mean perfect output.

The subsets of the shuffling framework need to be generated sequentially. There is a limit to how many computers the execution of the clone detection tool for the subsets can be distributed across before serial subset generation becomes a bottleneck. In contrast, the deterministic methods have marginal subset generation computation cost and could be distributed up to one computer per pair of partitions. So if a large cluster of computers is available, the deterministic method may be the better option.

In summary, the shuffling framework is a better option than the deterministic method when partial native recall is acceptable, and when computational resources are limited. The deterministic method is a better option when 100% native recall is preferred, and a large cluster is available. Our goal was to enable scalable clone detection with classical tools using a limited number of standard workstations. Our shuffling framework satisfies this use case, which is not satisfied by a deterministic method.

11. IJADATASET REVISITED

In Section 7, we used the original 'core' shuffling framework (blind partitioning shuffling algorithm) to evaluate the ~~ultra-large~~ IJaDataset. In Section 8, we discussed the deficiencies of the original technique, and in Section 9, we incrementally designed a technique that addresses these problems. We demonstrated the new technique's superiority by evaluating it using samples of the IJaDataset where it was possible to create gold standards for all of the tools.

In this section, we continue our primary experiment using our new inverted-index shuffling algorithm to detect clones in IJaDataset. We compare the two algorithms for their intended use case: big data clone detection. We limit our tool selection to Simian and NiCad. We include NiCad as the new algorithm worked best with it in the simulated experiments (Section 9) and because it is fast for function clone detection. We include Simian because the new algorithm showed the most conservative improvements with it and because it is the only tool we have a gold standard for. We omit Deckard because it requires long computation times, even for smaller datasets. We do not have the computational resources to dedicate to it. We omit iClones because it did not participate in the previous IJaDataset experiment, so we cannot compare the two versions of the framework with it.

Unfortunately, prohibitive memory and execution time requirements prevented us from building gold standards for IJaDataset for any of the tools except Simian. Simian required a rented Amazon EC2 instance with 64 GB of RAM and days of execution time to evaluate the IJaDataset. Simian is quite fast because it only considers type 1 and ~~2~~ clones. Renting this server for tools that have similar memory requirements, but much longer execution times, was financially prohibitive.

For these experiments, we executed the inverted-index shuffling algorithm for a 3-gram representation of the dataset. The similarity heuristic was parameterized to require selected file pairs to share three 3-grams. The inverted index was trimmed of any 3-grams appearing in more than 1000 files. These are the same settings used in the evaluation of the algorithm for the small test datasets. Because these settings produced good results in the test case (e.g., the framework achieved 98% total recall with NiCad), we are optimistic that they are good parameters for the IJaDataset. It took 12 h to hash the IJaDataset and 40 min to build the 3-grams. It took 15 min to build the index and 2.3 min to trim it. The hashed dataset only needs to be produced once and can be used with multiple executions of the shuffling framework with multiple subject tools. Changes to the dataset only require re-hashing of new or changed files.

As per the previous IJaDataset experiment, we used a maximum subset size of 50,000 files for Simian and a maximum subset size of 10,000 files for NiCad. While NiCad could handle the 50,000-file dataset used in the evaluation of the shuffling algorithm improvements, it does not reliably in the general case, which is why a smaller subset size is used.

DOI: 10.1002/smr

Recall that the inverted-index algorithm executes two rounds of detection subsets. In the first round, the dataset is fully partitioned into subsets using blind shuffling. The first round parallels the original 'blind' shuffling algorithm and ensures that the clone detector is exposed to all of the intra-file clones in the dataset. The number of subsets in the first round is equal to the size of the dataset divided by the subset size for the tool (rounded up). The second round of subsets is constructed using the algorithm's new selection criteria. Pairs of files in the index that share an *n*-gram are selected at random and added to the subset if they satisfy the similarity heuristic and have not been previously seen together. The second round can have any number of subsets.

11.1. Simian

Simian detects a very large number of clone pairs in IJaDataset, more than we can process on our hardware. As with our previous IJaDataset experiment, we instead measured our clone fragment recall heuristic. The framework's clone fragment recall for Simian is shown in Figure 19. For comparison, we also plot the fragment recall when the blind shuffling algorithm was used. For the first round (58 subsets), both algorithms use blind shuffling and obtain essentially identical recalls. Once the inverted index switches to its inter-file detection strategy, there is a huge difference in algorithm performance. The inverted-index algorithm obtains nearly the same fragment recall within 200 subsets as the blind algorithm does in 900 subsets. In this case, the inverted index reduces the amount of required work by nearly 80%. It is interesting to note that the inverted-index algorithm provides a very quick burst of recall growth across the initial subsets, but the rate of growth quickly diminishes. It may be possible that it is beginning to reach an asymptote. This is not very desirable and suggests that the framework may achieve a higher end recall with Simian with more relaxed file selection parameters (*n*-gram length, minimum similar *n*-grams, and inverted-index trim threshold). However, we had seen some strange behavior in previous experiments with Simian, for example, the sliding effect we had previously mentioned, where Simian was reporting the same clones repeatedly with small differences in start/end lines. If the sliding effect was more pronounced when the input size was larger (e.g., the creation of the gold standard vs. the detection of the subsets), it would cause a low total recall to be measured. For this reason, we also decided to simulate Simian's detection of the subsets.

Also plotted in Figure 19 is the fragment recall results of our simulation of Simian's execution for the subsets. The simulation assumed that a clone pair in Simian's gold standard was detected if the file(s) containing the clone were seen within the same subset. For measuring clone fragment recall, the cloned fragments of the detected clone pairs are also detected. In the simulated case, we see much higher cloned fragment recall. This tells us that Simian fails to report some of the clone pairs in its gold standard even when the files containing these clones are shuffled together. Simian may be reporting clones inconsistently for a particular pair of files based on what other files they are input

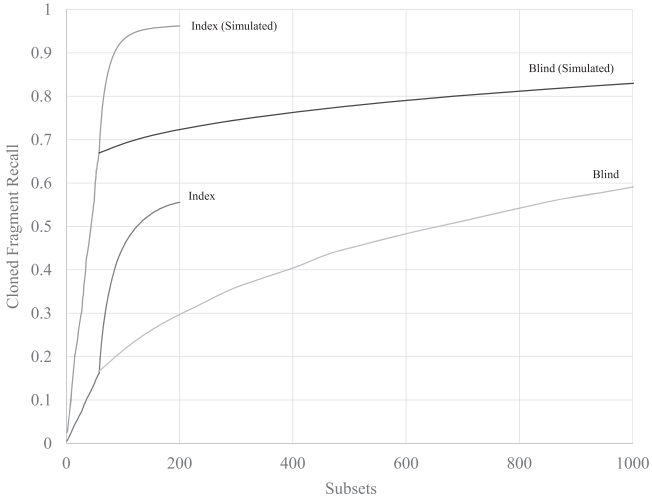


Figure 19. Index versus blind shuffling algorithm for IJaDataset using Simian.

DOI: 10.1002/smr

with. Or Simian's sliding effect defect may be more pronounced for a larger input size (e.g., large gold standard vs. small subset). The discrepancy between the non-simulated and simulated Simian subset detection occurs even within the first 58 subsets (round 1). During these subsets, both algorithms partition the entire dataset using blind shuffling. This suggests that Simian is not reporting many of the intra-file clones in its gold standard when it is analyzing the files in small subsets rather than all in one input. Possibly, the sliding effect is more pronounced for larger input, specifically reporting many slight variations on the same fragment as a large clone class. This is consistent with casual observations we have made of Simian's gold standard.

From our analysis of Simian, we wished to estimate the general performance of the shuffling framework with the inverted-index algorithm with any clone detection tool. We imagine that the performance lies somewhere between these two evaluations with Simian. The evaluation with real detection data underestimates our framework's performance as Simian is failing to detect, or is reporting differently, clones in Simian's gold standard that the shuffling framework has exposed Simian to. However, the simulation may be overestimating the recall. If Simian's sliding effect produced a large number of intra-file clones, it may be overwhelming the number of inter-file clones and boosting the recall within the 58 subsets higher than if Simian did not have this defect.

The framework was executed on a solid-state drive, which provided greatly improved execution time of both the shuffling framework and Simian. Building the detection subsets from their specification (i.e., assembling the files for detection) took less than 5 min per subset. Choosing the files for the subsets of round 1 (blind shuffling) took less than 1 s per subset. For round 2, subset generation time started at 30 s per subset and increased as more subsets were generated up to 44 min per subset by the 142th subset in round 2 (200th subset in total), with a total generation time of 24 h. From the study of the inverted-index algorithm in Section 9, we saw that when subset generation time had increased by two orders of magnitude, the framework had mostly exhausted its search space and reached its maximum total recall. Subset assembly took on average 1.33 min per subset. Simian's execution time per subset was 1.28 min on average, with a range of 7 s to 12 min and a total execution time of 5 h. For Simian, subset generation time exceeds execution time. This is expected as Simian's scalability limit for big data is not execution time, but its memory requirements.

11.2. NiCad

Like the previous IJaDataset experiment, we evaluated the shuffling framework's performance with NiCad by measuring the number of unique detected clone pairs and cloned fragments across the subsets. We could not measure total recall as NiCad cannot be scaled to IJaDataset even with extraordinary hardware. It has internal limitations that restrict the size of the input in terms of the number of source lines and the amount of cloned code. Even if these limitations are removed and given sufficient RAM, it could take months of execution time to produce the gold standard.

In Figure 20, we show the shuffling framework's cumulative detection of unique clone pairs across the subsets using NiCad. We also show the detection performance using the blind shuffling algorithm for comparison. The blind algorithm data are taken from the previous IJaDataset experiment (Section 7). The detection rounds for both algorithms are indicated by the circle markers.

For the first round, both algorithms use blind shuffling and have nearly identical clone pair detection performance. Once the index algorithm begins its second round, a considerably better clone pair detection performance is observed. The index algorithm is able to detect approximately the same number of clone pairs in 438 subsets as the blind algorithm does in 5780 subsets (20 'blind shuffling' rounds), a 92% reduction in subsets. The number of subsets needed to achieve the same result is reduced by a whole order of magnitude. This is a considerable decrease in the number of required clone detection tool executions.

In Figure 21, we show the shuffling framework's cumulative detection of unique clone fragments across the subsets using NiCad. Again, for the first round where both algorithms use blind shuffling, the detection performance is essentially identical. Like with the clone pairs, we see a large improvement in cloned fragment detection using the index algorithm over the blind algorithm. The index algorithm detected approximately the same number of cloned fragments within 299 subsets as the blind algorithm did in 5780 subsets, a 95% reduction in subsets. The index algorithm finds

DOI: 10.1002/smr

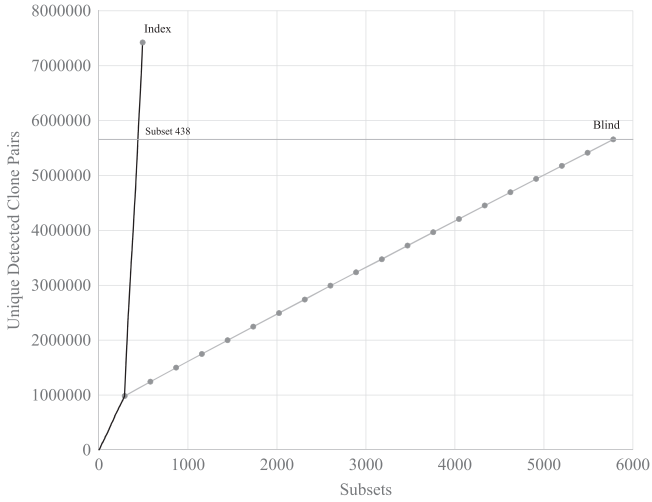


Figure 20. Index versus blind shuffling algorithm clone pair detection for IJaDataset with NiCad.

nearly double the cloned fragments within 488 subsets as the blind algorithm does in 5780 subsets. Considering only the clone pairs detected after the first round (where both use blind shuffling), the index algorithm detects 3.5× the clone pairs in 200 subsets as the blind algorithm does in 5493 subsets.

The improvement in cloned fragment detection with the index algorithm is larger than that of the improvement in clone pair detection. Unlike with the clone pairs, we see a noticeable decay in the growth of detected cloned fragments. The average number of new cloned fragments detected per subset is decreasing, suggesting that they are becoming rarer. This suggests that the cloned fragments are being found faster than the clone relationships between them. A transitive clone recovery technique could be used to recover these missing relationships without additional subsets. Because the index algorithm is detecting the cloned fragments much faster than the blind algorithm does, the transitive recovery technique would be even more valuable when used with the index approach. Before this is possible, an efficient and precise way to apply transitivity to type 3 clones, for which the validity of transitivity would need to be checked in each instance, needs to be devised.

The framework was executed on a solid-state drive, which greatly improved the execution time of the shuffling framework and NiCad. Building the subsets after their file contents had been chosen took 0.25 min on average. Choosing the files for the subsets of the first round (blind partitioning, subsets 1–258) took 60 ms on average. For round 2 (inverted index, subsets >248) and consistent with our framework improvement study, subset generation time started short (a few seconds) and

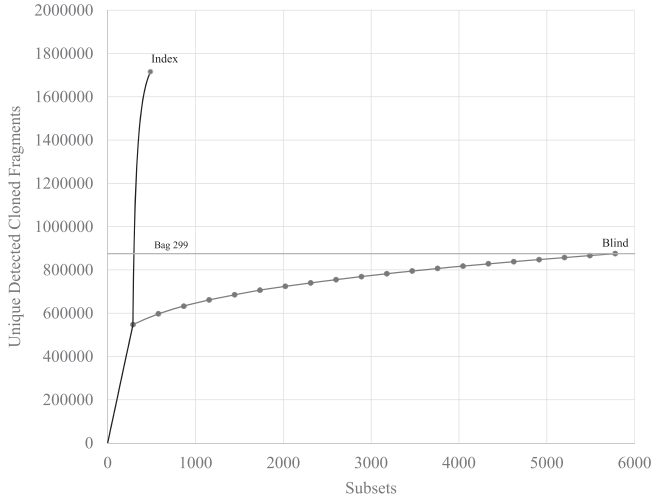


Figure 21. Index versus blind shuffling algorithm clone fragment detection for IJaDataset with NiCad.

DOI: 10.1002/smr

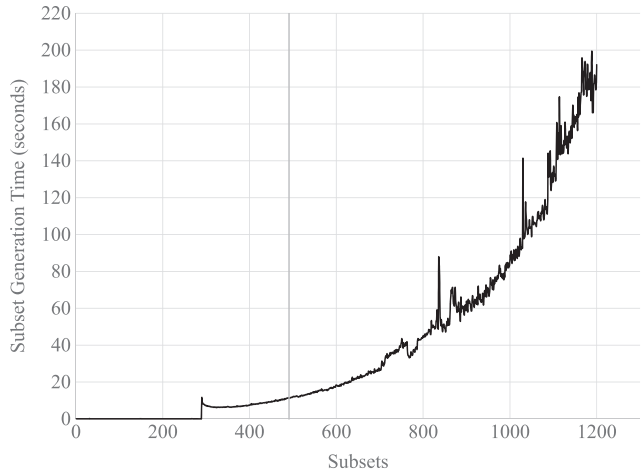


Figure 22. Subset generation time—10,000-file subsets of IJaDataset–NiCad.

grew as more subsets were generated to a couple of minutes by subset 1100. NiCad's execution time per subset was 3.5 min on average, with a range of 1.3–15.6 min.

Subset generation time is plotted in Figure 22. The gray line shows the subset we stopped executing NiCad at for this experiment. Between subset 259 (the start of the inverted-index algorithm) and subset 1200, the subset generation time increases by an order of magnitude. When we tested the inverted-index algorithm with NiCad and a dataset of 50,000 files, we found that total recall had reached 90% by the time that the subset generation time had increased by an order of magnitude. Because we used a similar ratio between subset size and dataset size in the test experiment as we have used in this IJaDataset experiment, perhaps the shuffling framework would achieve 90% total recall of NiCad's gold standard for the IJaDataset by subset 1200. Unfortunately, we cannot verify this as it is not practical to compute NiCad's gold standard for the IJaDataset.

With the inverted-index algorithm, the subsets must be generated serially, as the contents of a subset depend on the contents of previous subsets. This is a potential bottleneck if the execution of the tool for these subsets is distributed over a number of computers. In Figure 23, we plot the time at which each subset is ready for evaluation. This is the cumulative subset generation time versus subset. Alongside

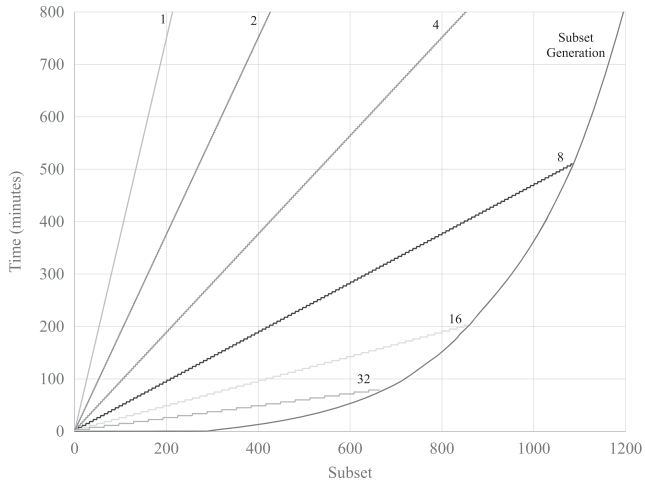


Figure 23. Subset Generation Bottleneck - Comparing the Time When a Subset is Ready for Evaluation Against the Time its Evaluation is Complete by Parallel Executions of NiCad for Various Compute Cluster Sizes (1–32).

DOI: 10.1002/smr

this, we plot the time at which NiCad's evaluation of a subset is complete when 1, 2, 4, 8, 16, or 32 computers are utilized. Time is counted in minutes from when the generation of the first subset of round 1 began. For this calculation, we considered a computer occupied for 3.75 min to evaluate a subset, which includes the average NiCad execution time for a 10,000-file subset, and the average time required to assemble a subset from its specification (a file list). We assume that IJaDataset is on each computer and that it takes negligible time for a subset specification to be sent to a computer. Because we are assuming a uniform NiCad execution time, we consider every nth subset to be sent to a specific computer, where n is the number of computers. For example, with four computers in a cluster, subsets 1, 5, 9, … go to computer 1; subsets 2, 6, and 10 go to computer 2; and so on. Therefore, NiCad's analysis of a subset is complete exactly 3.75 min after the later of the following: (1) the time the subset's generation was complete or (2) the time at which the computer finished analyzing its previous subset. Case (1) will only happen when a computer is waiting for its first subset and once generation time becomes a bottleneck and the computer is idly waiting for its next subset to be generated.

From this distributed execution estimate, we do not see one, two, or four computers becoming bottlenecked by subset generation within the 1200 subsets we generated. Eight computers become bottlenecked after 1080 subsets, after which at least one computer is idle. This occurs at subset 854 with 16 computers and subset 664 with 32 computers. The intention of this framework was to enable the scaling of classical tools on standard hardware. This plot shows us that the shuffling framework's subset generation time will not bottleneck a budget compute cluster of two to four computers. The bottleneck may occur at a later subset when the framework is used with other tools, or even different configurations of NiCad, that require longer execution times. For this experiment, we executed NiCad in its most basic configuration. Because NiCad was only looking for function granularity clones and did not perform any normalization beyond pretty printing, its execution time for the 10,000-file subsets was quite fast for type 3 detection. Tools that look for clones at lower granularities will likely have longer execution time. NiCad's execution time is longer when its advanced features are enabled.

The bottleneck could be overcome by generating multiple subsets simultaneously when a computer in the cluster is idle. When determining if a randomly selected pair of files (which satisfy the similarity heuristics) is unseen, the algorithm would consult the contents of the previously generated subsets, but not the contents of other subsets being generated simultaneously. As such, some of the same file pairs may be selected for subsets generated at the same time. However, the probability of this would be low unless total recall was close to 100%. This technique would ensure that all the computers in the cluster are continuously utilized. We will explore such a scheme as part of our future work toward a publicly released tool version of this framework.

11.3. Summary

In summary, we experienced considerable gains in detection performance with the inverted-index shuffling algorithm over the blind shuffling algorithm when evaluating the IJaDataset using Simian and NiCad. With Simian, we found that the index algorithm allows a higher cloned fragment recall to be obtained with fewer subsets. With NiCad, we found that the index algorithm was able to match the blind algorithm using an order of magnitude fewer subsets. From these results, we conclude that the inverted-index algorithm greatly exceeds the performance of the blind shuffling algorithm (RQ#5).

12. CONCLUSION

In this research, we presented and demonstrated the shuffling framework for scaling classical clone detection tools to big data on standard consumer-level (i.e., affordable) workstation-class hardware. The shuffling framework scales classical tools by executing them for non-deterministically chosen subsets of a big data source dataset. We began with the version of the shuffling framework we proposed in previous work [1, 2], which we termed the 'core shuffling framework', which used the 'blind partitioning shuffling algorithm'. We evaluated this version of the framework using ordinary-

DOI: 10.1002/smr

sized systems (for comparison against gold standards) and for its application to real big data (IJaDataset 2.0). While this version of the framework successfully scaled the classical tools to big data, the execution time required to obtain a satisfactory ratio of a tool's native recall was still high. We used these experiments to identify the deficiencies in the approach. Specifically, the blind partitioning algorithm did not prevent the same files from being randomly shuffled together repeatedly, and it did not consider the similarity of the files it was shuffling together.

Considering these deficiencies, we iteratively improved the shuffling framework by modifying the original shuffling algorithm. We explored methods of tracking files that have been seen by the tool previously (to prevent shuffling them together repeatedly), as well as *n*-gram and inverted index-based file-similarity heuristics (to prevent shuffling together dissimilar files). We evaluated the improvements using a sample of the IJaDataset small enough to evaluate the subject clone detectors' gold standards. We evaluated the improvements using the same subset size to dataset size as used in the big data case with the IJaDataset. We termed our final algorithm the 'inverted-index shuffling algorithm'. We found from our evaluations that this algorithm was able to scale clone detectors to big data while capturing up to 90–95% of the a clone detector's native recall without sacrificing its precision. We then applied our new algorithm to the big data IJaDataset and found that it was able to capture the detection performance of our original 'blind partitioning shuffling algorithm' using 90% fewer subsets of the IJaDataset, thereby improving our framework's scalability by an order of magnitude.

Using our approach, classical clone detectors can be used to detect clones in big data on commodity hardware. Researchers and developers can use their familiar, available, proven, and well-understood classical tools to build clone corpora for ~~ultra-large~~ inter-project software datasets. These corpora may be used to study developer behavior within a corporation or globally (open source). Duplicated engineering efforts in open source or within a corporation can be reduced by extracting the duplication found into new software libraries. Large corpora can be used within Internet-scale clone search to provide API recommendation and usage support. Our approach comes at the cost of a fraction of a tool's native recall. However, a good clone corpus is built using multiple scalable and classical tools. In this intended use case, lower native recall is made up for by the consultation of multiple and varied clone detectors.

13. FUTURE WORK

Our primary goal in future work is to create a tool version of the improved shuffling framework for use by researchers and professionals. In this research, we used a prototype version of the shuffling framework to analyze its effectiveness and performance. A tool version would automate the process and provide ease of use to the user. At most, a framework user would need to implement a standard communication protocol between their clone detection tool of choice and the framework. The tool would provide flexibility in executing the tool for the subsets in series, in parallel, or in a distributed fashion. For the implementation of a general shuffling framework tool, we would focus on reducing the computation time required to choose the contents of the subsets of the second detection round and the computation time required to hash the dataset.

We plan to revisit the transitive clone recovery technique. We demonstrated that the recovery technique was effective in recovering the missed clone relationships between detected cloned fragments. We also found that the framework locates the cloned fragments in big data faster than the clone relationships between them. Therefore, the best time to execute clone recovery would be when the framework reaches a point of diminishing returns in the number of new cloned fragments detected per subset. To be practical, the clone recovery technique must discover the remaining clone relationships between the known cloned fragments in less computation time than executing the framework and classical tool for additional subsets. The recovery technique also needs to maintain high precision. While transitivity always holds for clones of types 1 and 2, it may not for type 3. Efficiently checking for type 3 transitivity is a key factor in a recovery algorithm's performance. At the very least, the recovery algorithm must meet the precision of the classical tool being used.

DOI: 10.1002/smr

We are also interested in creating a clone detection tool benchmark for detection in big data. This would allow us to better measure the performance of our shuffling framework for a variety of classical clone detection tools. Using this benchmark, we could also compare the performance of our framework against the other scalability tools and techniques listed in Section 2. Such a benchmark would improve community confidence in scalable clone detection techniques and encourage continued improvements in scalable clone detector recall and precision.

ACKNOWLEDGEMENTS

We thank the developers of the tools used in this study, as well as Dr. Rilling and the co-authors of our earlier publications on the shuffling framework [1] and IJaDataset [4].

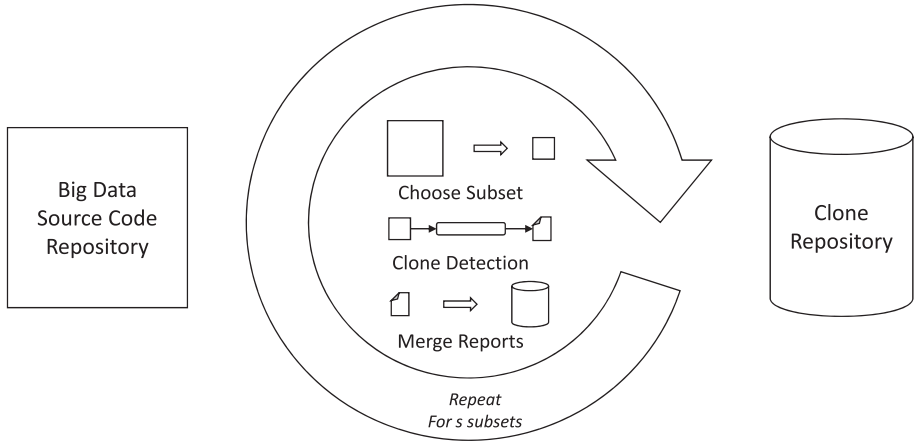
REFERENCES

1. Keivanloo I, Roy CK, Rilling J, Charland P. Shuffling and randomization for scalable source code clone detection. *6th International Workshop on Software Clones (IWSC)*: Zurich, Switzerland, 2012; 82–83.
2. Svajlenko J, Keivanloo I, Roy CK. Scaling classical clone detection tools for ultra-large datasets: an exploratory study. *Proceedings of the ICSE 7th International Workshop on Software Clones (IWSC)*: San Francisco, CA, USA, 2013; 16–22.
3. Keivanloo I, Forbes C, Rilling J. Similarity search plug-in: clone detection meets internet-scale code search. *ICSE Workshop on Search-Driven Development—Users, Infrastructure, Tools and Evaluation (SUITE)*: Zurich, Switerland, 2012; 21–22.
4. Keivanloo I, Forbes C, Hmood A, Erfani M, Neal C, Peristerakis G, Rilling J. A linked data platform for mining software repositories. *9th IEEE Working Conference on Mining Software Repositories (MSR)*: Vancouver, BC, Canada, 2012; 32–35.
5. Jiang L, Misherghi G, Su Z, Glondu S. Deckard: scalable and accurate tree-based detection of code clones. *29th International Conference on Software Engineering (ICSE)*: Minneapolis, MN, USA, 2007; 96–105.
6. Roy CK, Cordy JR. Nicad: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. *The 16th IEEE International Conference on Program Comprehension (ICPC)*: Amsterdam, The Netherlands, 2008; 172–181.
7. Gode N, Koschke R. Incremental clone detection. *13th European Conference on Software Maintenance and Reengineering (CSMR'09)*: Kaiserslautern, Germany, 2009; 219–228.
8. Harris S. Simian—similarity analyzer. 2011. Available from: http://www.harukizaemon.com/simian/ [Accessed 1 January 2013].
9. Uddin MS, Roy CK, Schneider KA, Hindle A. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. *18th Working Conference on Reverse Engineering (WCRE)*: Limerick, Ireland, 2011; 13–22.
10. Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
11. Koschke R. Large-scale inter-system clone detection using suffix trees. *16th European Conference on Software Maintenance and Reengineering (CSMR)*: Szeged, Hungary, 2012; 309–318.
12. Ishihara T, Hotta K, Higo Y, Igaki H, Kusumoto S. Inter-project functional clone detection toward building libraries—an empirical study on 13,000 projects. *19th Working Conference on Reverse Engineering (WCRE)*: Kingston, ON, Canada, 2012; 387–391.
13. Sajnani H, Ossher J, Lopes C. Parallel code clone detection using MapReduce. *IEEE 20th International Conference on Program Comprehension (ICPC)*: Passau, Germany, 2012; 261–262.
14. Livieri S, Higo Y, Matushita M, Inoue K. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. *29th International Conference on Software Engineering (ICSE)*: Minneapolis, MN, USA, 2007; 106–115.
15. Schwarz N, Lungu M, Robbes R. On how often code is cloned across repositories. *34th International Conference on Software Engineering (ICSE)*: Zurich, Switzerland, 2012; 1289–1292.
16. Ossher J, Sajnani H, Lopes C. File cloning in open source java projects: the good, the bad, and the ugly. *27th IEEE International Conference on Software Maintenance (ICSM)*: Williamsburg, VA, USA, 2011; 283–292.
17. Roy CK, Cordy JR. *A Survey on Software Clone Detection Research*. School of Computing TR 2007-541, Queens University: Kingston, ON, Canada, 2007; 115.
18. Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 2007; **33**(9):577–591.
19. Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Science of Computer Programming* 2009; **74**(7):470–495.
20. Ambient Software Evolution Group. SECold IJaDataset 2.0. January 2013. Available from: http://secold.org/projects/seclone [Accessed 1 January 2013].

DOI: 10.1002/smr

Big data clone detection using classical detectors: an exploratory study

Jeffrey Svajlenko, Iman Keivanloo and Chanchal K. Roy



Big data clone detection across tens of thousands of software systems has several applications, including API usage recommendation, code completion, and search driven development. However, the state-of-the-art tools are designed to scale to only single software systems. We develop a scalability heuristic that scales these classical tools to tens of thousands of software systems using commodity hardware and evaluate its performance experimentally.