# Clone Detection on Large Scala Codebases

Wahidur Rahman*, Yisen Xu†, Fan Pu†, Jifeng Xuan†, Xiangyang Jia†,
Michail Basios‡, Leslie Kanthan‡, Lingbo Li‡, Fan Wu‡ and Baowen Xu§
*Imperial College London, London, United Kingdom
w.rahman17@imperial.ac.uk
†Wuhan University, Wuhan, China
{xuyisen,fan_pu,jxuan,jxy}@whu.edu.cn
‡Turing Intelligence Technology, London, United Kingdom
{mike, leslie, lingbo, fan}@turintech.ai
§Nanjing University, Nanjing, China
bwxu@nju.edu.cn

***Abstract*—Code clones are identical or similar code segments. The wide existence of code clones can increase the cost of maintenance and jeopardise the quality of software. The research community has developed many techniques to detect code clones, however, there is little evidence of how these techniques may perform in industrial use cases. In this paper, we aim to uncover the differences when such techniques are applied in industrial use cases. We conducted large scale experimental research on the performance of two state-of-the-art code clone detection techniques, *SourcererCC* and *AutoenCODE*, on both open source projects and an industrial project written in the Scala language. Our results reveal that both algorithms perform differently on the industrial project, with the largest drop in precision being 30.7%, and the largest increase in recall being 32.4%. By manually labelling samples of the industrial project by its developers, we discovered that there are substantially less Type-3 clones in the aforementioned project than that in the open source projects.**

***Index Terms*—Clone Detection, Scala Language**

## I. Introduction

With more and more software source code constantly being published on open source platforms such as *GitHub*, code reuses or code clones are common to many repositories [1], [2]. Code clones are segments of source code that are identical or similar in syntax or semantics [3], [4]. Developers often create code clones via copying existing code and pasting with or without modifications to reduce the development time and costs. However, code clones were shown to increase software maintenance costs, since software bugs can easily propagate via code cloning and inconsistent fixing of these bugs may induce undefined behaviour [5]. While completely avoiding code clones is impractical, researchers have proposed different detection techniques to identify code clones in existing codebases to reduce the risk of code clones.

Existing clone detection techniques can be generally categorised according to the code representation used in the algorithm: text-based [6], lexical- or token-based [7], [8], graph-based [9], abstract syntax tree-based [10], [11], bytecode-based [12], [13], or their combinations [14], [15]. Among different clone detection techniques, token-based algorithms have shown promising results whilst being scalable to large datasets [16]. One state-of-the-art token-based algorithm is SourcererCC [8], which incorporates several optimisation techniques that dramatically improve its speed. Meanwhile, with growing attention to deep learning, White et al. and Tufano et al. introduced and improved a deep learning technique, AutoenCODE for clone detection [17], [18], which outperforms many previous algorithms.

Despite promising reports, the performance of these state-of-the-art algorithms have only been applied to open source codebases, where code reuse and foraging is a common practice [1]. In contrast, to protect their source code from leaking, many commercial organisations allow limited or no access to public open source platforms. This makes code reuse from external sources difficult or even impossible. In such a closed programming environment, detected code clones are likely to be different from those in an open environment where there are rich sources for code cloning. Therefore, investigation is required for ascertaining whether the performance of clone detection algorithms may change when applied to closed codebases.

In this paper, we empirically investigate the performance of clone detection on an industrial project with over 4 million lines of code (LoC) written in Scala, a functional programming language. We aim to examine two state-of-the-art clone detection algorithms, namely SourcererCC and AutoenCODE, on a private industrial project and determine the potential differences in performance when applying them to open source projects. Both algorithms are adapted to accept programs written in Scala as the industrial codebase is written in Scala. The source code in the industrial codebase was created in an environment where downloading code from external sources was strictly forbidden. Such programming environment is not uncommon in many industrial organisations, therefore, the private industrial codebase we studied in this paper can be an example of how clone detection techniques may perform in industrial use cases in general. To compare the performance of the algorithms on industrial and open source codebases, we apply the algorithms to the industrial codebase and the top 20 Scala projects on *Github*, ranked according to the number of stars of the projects. Samples of code clones are manually labelled by human developers, and precision and recall metrics are calculated as measurements of the algorithms' performance.

Experimental results show that, when applied to the industrial codebase, both algorithms show different degrees of

degradation in precision, with the biggest drop from 95.1% to 64.4%, but the recall can improve as much as 32.4%. We also observed a substantially lower proportion of Type-3 clones (definitions of different types are introduced in Section II) in the industrial codebase, and a higher proportion of the samples are not considered as clones. Upon discussion with industrial developers, we discovered that many developers may consider unrefactorable clones as having little or no value to them and may be reluctant to classify such code segments as clones.

## II. Background

### A. Scala Language

Scala is a general purpose programming language that makes use of both, object-oriented and functional programming paradigms. It is closely related to the Java programming language as it compiles to java bytecode and provides interoperability with packages written in Java, but also provides functional programming features such as currying, higher order functions, type inference and pattern matching. While evaluation of clone detection techniques on the Java language exists in great number in the literature, evaluation on Scala or other functional languages is lacking.

Listing 1: Example of two similar methods in Scala language

```
def secondElementIfArray(x: Any) = x match {
  case Array(_, a, _*) => a
  case _ => "default"
}

def nameIfDog(x: Any) = x match {
  case Dog(a) => a
  case _ => "default"
}
```

An example of a syntactically similar method pair in Scala language is shown in Listing 1. In the example, the first method checks whether the input is an array with at least two elements and returns the second element of the array. The second method checks whether the input is of the class `Dog`, where `Dog(name: String)` is the constructor of the class. It returns its member variable `Dog.name` if it is the matching class. Though these two methods implement quite different functionalities, due to the *pattern matching* feature of Scala, the two methods appear similar in syntax. Therefore, these two methods can be identified as clones by clone detection algorithms. However, clone detection for Scala programs may be distinct from that for other languages. For instance, if the two methods in Listing 1 are implemented in Java, one may use size checking on an array and an array elements' access, while the other requires class type checking and the *getter* function of a member variable. Therefore, they may appear different and subsequently make a clone detection algorithm identify them as non-clones.

### B. Types of Code Clones

We follow the widely accepted definitions of different types of clones [3], [4], [8]:

**Type-1**. Identical code fragments except for differences in whitespace, comments and layout.

**Type-2**. Identical code fragments except for differences in identifier names and literal values, in addition to Type-1 clone differences.

**Type-3**. Syntactically similar code fragments that contain added, modified and/or removed code statements with respect to each other, in addition to Type-1 and Type-2 clone differences.

**Type-4**. Syntactically different code fragments, but are semantically similar in terms of their implemented functionalities.

### C. SourcererCC and AutoenCODE

In this paper, we investigate the performance of two state-of-the-art clone detection techniques; SourcererCC is a token-based algorithm and AutoenCODE incorporates deep learning with different code representations [8], [18].

SourcererCC is a state-of-the-art token-based clone detection algorithm proposed by Sajnani et al. [8]. It represents a code fragment as a bag of tokens (tokens that appear in the fragment and their frequencies). The clone detection criterion is deterministic and based on the degree of overlap between the bags of tokens from two code fragments, and a percentage similarity threshold is used as a cutoff when deciding whether they are clones. This simple idea requires pairwise comparisons of all methods to assess the degree of overlap and is therefore $O(n^2)$ complexity and hard to scale to large real-world projects. SourcererCC overcomes this by exploiting two properties of token features, which define the upper and lower bounds of similarity if only a subset of tokens is seen. Through the creation of a memory efficient partial index for candidates that satisfy these properties, and repetitively updating the upper and lower bounds of similarity measure as more tokens are seen, it is then able to eliminate the majority of method pairs as early as possible with certainty. With such optimisation, the number of method pairs requiring a full comparison is greatly reduced, thus the speed of the algorithm is dramatically increased.

AutoenCODE is a state-of-the-art clone detection algorithm based on deep learning, a neural network based technique to minimise the need for manual feature engineering. It works by generating sentence embeddings using deep learning on four different representations of code fragments: *Identifier*, Abstract Syntax Tree (*AST*), compiled bytecode, and Control Flow Graph (*CFG*). The first two representations can be obtained from the source code, and the latter two are usually obtained from compiled binary code. AutoenCODE works primarily in four stages. In the first stage, code representations are extracted from both source code and binary code, and word vectors are generated at the required code granularity level (in our case, we require a vector for each method). In the second stage, word embeddings for each word in the word vector of a method are learnt using a Recurrent Neural Network. In the third stage a sentence embedding is learnt for each method with a Recursive Auto Encoder [19], using the word embeddings from the second stage. In the final stage, euclidean distances between sentence embeddings are computed and a distance threshold is used to determine which methods are clones. The algorithm detects clones using each code representation independently, and the results can be combined in different ways to form the final result.

## III. Experiment Design

In this section, we describe in details the datasets, the experiment procedure, and the Research Questions (RQs).

### A. Datasets

In order to evaluate and compare the performance of clone detection algorithms on open source and industrial codebases, we use two separate datasets in our experiments: an open source dataset and a private industrial dataset.

The open source dataset is composed of the top 20 most popular Scala projects (that received the most stars) on *GitHub*. Since more popular projects tend to have more developers contributing to the project, the collaborative coding practice should be the closest to that of an industrial project. A summary of the 20 Scala projects, including their lines of code and number of methods, is outlined in Table I. The numbers of lines of code are counted after comments and empty lines are removed, and only the methods with a minimum of 10 lines are considered, which is a common practice for clone detection [3], [8]. The project sizes vary from $2,091$ to $305,276$ lines of code, and the numbers of methods vary from $30$ to $5,256$.

The industrial project is obtained via a collaboration with an industry organisation, which wishes to remain anonymous. The project consists of more than 4 million of lines of code, and is written and maintained by hundreds of developers within the company. Due to the specific domain of the company, developers are given limited access to open source platforms. For instance, open platforms such as *SourceForge* and *BitBucket* are completely blocked within the company. *GitHub* can be viewed, but downloading open source projects without approval is strictly forbidden. Similar requirements can be found in many other private companies to protect the proprietary source code [20], [21]. Code written in such an environment is to our interests as it is difficult or even impossible to copy open source contents and the code cloning behaviours can be different. This may lead to a different pattern of code clones in the codebase, and therefore different performance of clone detection algorithms. To gain access to this private codebase, some of the authors worked within the company for a period of time to conduct the experiments.

TABLE I: Projects summary.

| Project | Method Count | Lines of Code |
|---|---|---|
| scala-2.13.x | 5,256 | 305,276 |
| playframework | 894 | 69,653 |
| gitbucket | 371 | 22,754 |
| finagle | 1,663 | 130,195 |
| kafka-manager | 238 | 13,310 |
| lila | 1,161 | 77,669 |
| bfg-repo-cleaner | 30 | 2,091 |
| fpinscala | 67 | 7,378 |
| gatling | 432 | 33,375 |
| scalaz-series-7.3.x | 435 | 44,237 |
| incubator-openwhisk | 591 | 52,938 |
| sbt | 865 | 44,350 |
| scala-js | 3,373 | 133,952 |
| scala-native | 1,610 | 103,170 |
| dotty | 4,310 | 284,081 |
| scalding | 745 | 48,440 |
| BigDL | 2,376 | 163,881 |
| breeze | 1,244 | 48,271 |
| shapeless | 604 | 30,491 |
| spray | 380 | 33,167 |
| **Open Source Total** | 26,645 | 1,648,679 |
| **Industrial Project** | 69,533 | 4,051,596 |

### B. Experiment Procedure

Our experiments consist of three steps: data filtering, data labelling, and clone detection for both open source projects and the industrial project. We describe the details of each step as follows.

*1) Data Filtering:* Instead of relying on other clone detection tools as oracle [22], we choose to manually label the data as our ground truth. The benefit of manual labelling is not just better accuracy, but also, for the industrial dataset, the developer labelled data can better represent the industrial viewpoint on what Type-3 and Type-4 clones are. As the number of method pairs grows quadratically with the number of methods, it is impractical to label all method pairs. Therefore, we use SourcererCC to filter method pairs such that we can focus on the pairs that are more likely to be clones. A 70% similarity threshold was given to the SourcererCC algorithm in the data filtering stage. Such threshold was chosen according to our observation that method pairs below this threshold are very rarely to be clones, therefore can be regarded as non-clones without affecting our results.

*2) Data Labelling:* For different datasets, similar approaches are used for data labelling. For the industrial dataset, we asked the developers who have contributed to the codebase to voluntarily label the data. We sent out the invitation of labelling to a group of 705 developers, and 67 developers responded to the invitation by labelling at least one of the method pairs. To make the labelling process simple and convenient, we created a web-based GUI within the company's system to show a randomly picked method pair to the participant, along with the definition of the four different types of clones and their corresponding examples. The participant is asked to label the method pair as which type of clone, or not a clone, based on his/her best knowledge. Owing to the timing and the voluntary nature of the data labelling, we obtained 201 labelled method pairs that at least two developers agreed on the label.

For the open source dataset, two of the authors labelled the data separately, when there was disagreement, another author would make a decision between the two different labels. All of the three authors who labelled the open source dataset had been working intensively on the adaptation of the clone detection algorithms to Scala language, therefore, all three authors understood Scala programs and labelled the data with confidence.

*3) Clone Detection:* To evaluate the performance of SourcererCC and AutoenCODE, we run both algorithms on the industrial codebase and open source codebases. In this paper, we only use two code representation for AutoenCODE: *Identifier* (leaf) and *AST* (path), as opposed to using all four

representations in the original publication [8]. This is because the other two representations require the binary code after compilation, which is forbidden for the industrial codebase; meanwhile, our preliminary experiments showed that the other two representations did not work well for Scala language. The results of both industrial and open source datasets are then validated by the labelled data. We calculated precision and recall metrics to compare the performance of the algorithms.

### C. Experimental Setup

For SOURCERERCC on both industrial and open source datasets, a 90% similarity threshold is used to identify code clones. For AUTOENCODE, we keep the default parameters recommended by the original authors.

The machine specifications for running the clone detection algorithms differ between the open source and industrial projects due to the company regulations regarding the use and location of the source code. Experiments on the industrial codebase were run on a virtual machine with six cores of Intel Xeon CPU and 64GB of memory; Experiments on the open source projects were run on Google Cloud instances with eight cores of virtual CPU and 30GB of memory.

**Implementation**. We re-implemented variants of algorithms SOURCERERCC and AUTOENCODE to support Scala language. For SOURCERERCC, we re-implemented the algorithm fully in Scala with the native *scala.meta* library for parsing Scala code. For AUTOENCODE, we also use *scala.meta* library to parse Scala code and extract tokens for the *Identifier* and *AST* representations. For the rest of the algorithm, we use the implementation from the original authors. Our implementation of SOURCERERCC and AUTOENCODE can be found on *GitHub*[1][2].

### D. Research Questions

In this paper, we aim to answer the following RQs. We explained the rationals behind each question as follows.

**RQ1** What is the performance of SOURCERERCC and AUTOENCODE on open source codebases after adapted for Scala Language?

This RQ is to understand the baseline of the performance of clone detection algorithms under investigation. Specifically, we are interested in whether an algorithm performs as good on Scala language as they do with other languages studied by previous researchers. We answer this question by measuring precision and recall metrics of both algorithms on 20 open source Scala projects. The execution time of the algorithms on each project is recorded to evaluate the scalability of the algorithms.

**RQ2** Is there any difference in the performance of the clone detection algorithms when they are applied on open source codebase and private industrial codebase?

We pose this RQ to understand whether the state-of-the-art clone detection algorithms perform differently on private industrial codebase. Using the data labelled by human developers, we calculate the precision and recall metrics of both algorithms, and compare them with their respective performances on open source datasets.

[1]https://GitHub.com/Wahidur-Rahman/scala_sourcererCC
[2]https://GitHub.com/Wahidur-Rahman/scala_autoencode

## IV. RESULTS

### A. RQ1. Results on Open Source Benchmarks

The first research question queries the performance of SOURCERERCC and AUTOENCODE on open source Scala projects. We assess the results of both algorithms on the 20 open source benchmarks listed in Table I.

To evaluate the performance of the algorithms, we manually labelled 1000 random samples of method pairs after the data filtering process. Using the labelled data and the result from both clone detection algorithms, we can draw the Confusion matrix to understand its performance (Table II). From the table, we can calculate *precision* and *recall* measurements of the algorithms. For instance, the *precision* and *recall* of SOURCERERCC are calculated as:

$$precision = \frac{TP}{TP + FP} = \frac{247}{247 + 1} = 99.6\%$$

$$recall = \frac{TP}{TP + FN} = \frac{247}{247 + 616} = 28.6\%$$

where $TP$, $FP$, $FN$ are the counts of True Positive, False Positive, and False Negative respectively.

Precision and recall values are calculated for each algorithm, and are summarised in Table III. According to the table, we can see that both algorithms show a high level of precision, with SOURCERERCC topping the table with 99.6% of precision. The precision of the SOURCERERCC algorithm is in keeping with that seen by Sajnani et al. [8], where they calculated a precision of 91% on a sample of clones from *BigCloneBench* [23]. For the AUTOENCODE algorithm, the precision values are as well similar to that reported by Tufano et al. [18]. According to their results, AUTOENCODE achieved 100% and 96% precision for *Identifier* and *AST* representations respectively, whereas it is 98.3% and 95.1% respectively in our findings. The recall values cannot be compared with previous studies directly since we used different means to obtain the "ground truth" of False Negatives.

we plot the execution time for all open source projects against their lines of code and number of methods in Figure 1 and Figure 2 respectively. The execution time for each algorithm uses a different scale on the y axis in the Figures.

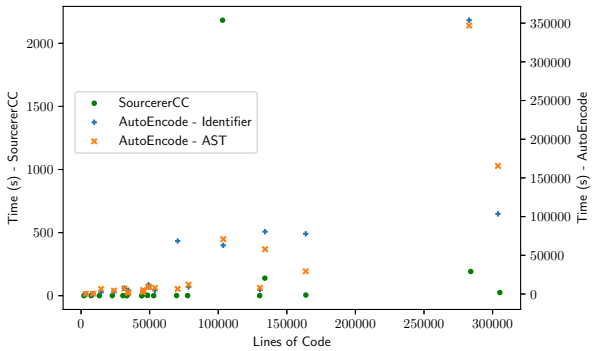


Fig. 1: Execution time as the Lines of Code increases for the open source benchmarks

The execution time of SOURCERERCC falls mostly in the range of 0.087 to 192 seconds, with an outlier of 2183 seconds

TABLE II: Confusion matrices of SourcererCC and AutoenCODE on the open source projects and the Industrial project. The *combination* columns are formed using the union of the Identifier and AST representation clones from the AutoenCODE algorithm.

| Open Source projects | | Truth (From Authors) | |
|---|---|---|---|
| | | **Clone** | **Not Clone** |
| SourcererCC identification | **Clone** | 247 | 1 |
| | **Not Clone** | 616 | 136 |
| AutoenCODE *(Identifier)* | **Clone** | 57 | 1 |
| | **Not Clone** | 806 | 136 |
| AutoenCODE *(AST)* | **Clone** | 117 | 6 |
| | **Not Clone** | 746 | 131 |
| AutoenCODE (Combination) | **Clone** | 139 | 7 |
| | **Not Clone** | 724 | 130 |

| Industrial Project | | Truth (From Developers) | |
|---|---|---|---|
| | | **Clone** | **Not Clone** |
| SourcererCC identification | **Clone** | 35 | 10 |
| | **Not Clone** | 28 | 128 |
| AutoenCODE *(Identifier)* | **Clone** | 13 | 2 |
| | **Not Clone** | 50 | 136 |
| AutoenCODE *(AST)* | **Clone** | 29 | 16 |
| | **Not Clone** | 34 | 122 |
| AutoenCODE *(Combination)* | **Clone** | 30 | 16 |
| | **Not Clone** | 33 | 122 |

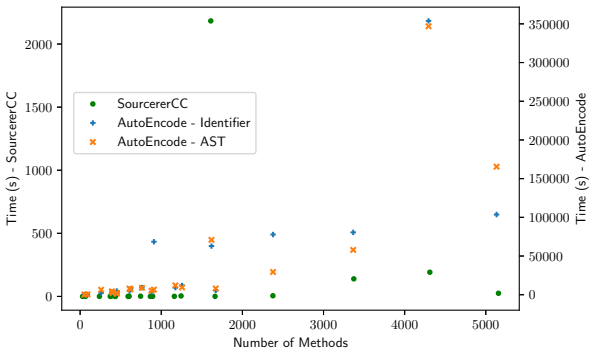


Fig. 2: Execution time as the Number of Methods increases for the open source benchmarks

TABLE III: Precision and recall metrics for AutoenCODE and SourcererCC on the open source and industrial projects

| | Open Source projects | | Industrial project | |
|---|---|---|---|---|
| | Precision | Recall | Precision | Recall |
| SourcererCC | 99.6% | 28.6% | 77.8% | 55.6% |
| AutoenCODE (*Identifier*) | 98.3% | 6.6% | 86.7% | 20.6% |
| AutoenCODE (*AST*) | 95.1% | 13.6% | 64.4% | 46.0% |
| AutoenCODE (Combination) | 95.2% | 16.1% | 65.2% | 47.6% |

observed on project *scala-native*. On the other hand, the execution time of AutoenCODE is generally 3 degrees of magnitude larger, ranging from 4.5 to 5781 minutes. These numbers are consistent with the execution time reported by both algorithms' original authors, where SourcererCC took a few seconds for projects in the order of $10^6$ lines of code, and AutoenCODE could take several hours.

**Summary**. Both SourcererCC and AutoenCODE achieve similar precision measurements on open source Scala projects, indicating that both algorithms retain performance when applied to functional languages. Though we use our own implementation of both algorithms, the execution time of both algorithms is consistent with their original authors. The timing of algorithm AutoenCODE can be up to 3 degrees of magnitude longer than SourcererCC.

### B. RQ2. Performance Comparisons

In this section, we assess the performance of the two algorithms on the industrial project, in terms of precision and recall metrics, and compare with their performance on open source projects. A summary of precision and recall values for both the industrial project and open source project can be found in Table III.

For the SourcererCC algorithm, the precision measurements drop on the industrial project, compared with the performance on Scala open source projects, however, the recall measurements increase. Precision drops from 99.6% to 77.8%, a difference of 21.8%, but recall increases from 28.6% to 55.6%, a difference of 27%. For the AutoenCODE algorithm, similar results can be observed. Precision measurements drop for all representations, with the biggest drop observed from 95.1% to 64.4% on *AST* representation, a difference of 30.7%. On the other hand, recall measurements increase for all representations, with the biggest increase from 13.6% to 46% on *AST* representation, a difference of 32.4%.

**Summary**. When both algorithms are applied to the industrial project, their performance changed substantially, with decreased precision and increased recall. The precision can drop as much as 30.7%, while the recall may increase as much as 32.4%.

### C. Discussion

In order to understand why the performance of the algorithm is different on the industrial project, we summarise the distribution of different types of clones in Table IV.

It is noticeable that there are differences in the distribution of the types of clones sampled from the open source project and the industrial project. Firstly, Type-2 clones constitute 11.8% of the samples from open source projects, while this number is only 7.5% for the industrial project. This can be due to the closed programming environment where the industrial project was created and maintained. Due to the limited access to other code sources, direct copying and pasting code segments happens less often, which may contribute to the less prevalence of Type-2 clones.

TABLE IV: Distribution of different types of clones

| Open Source projects | Type 1 | Type 2 | Type 3 | Type 4 | Not a Clone | Total |
|---|---|---|---|---|---|---|
| Open Source projects | 53 | 118 | 641 | 51 | 137 | 1000 |
| Open Source projects (%) | 5.3 | 11.8 | 64.1 | 5.1 | 13.7 | 100 |
| Industrial project | 11 | 15 | 28 | 9 | 138 | 201 |
| Industrial project (%) | 5.5 | 7.5 | 13.9 | 4.5 | 68.7 | 100 |

However, the differences in clone detection performance are more likely to be the direct result of much less Type-3 clones and much more non-clones in the industrial project sample. According to Table IV, only 13.9% of the industrial samples are Type-3 clones, while it is 64.1% in open source samples.

On the other hand, 68.7% of the industrial samples are not a clone, while this number is only 13.7% in open source samples. Upon discussion with some developers who labelled the industrial samples, we discovered that their views of what should be a clone might be different from the academic standards. For industrial use cases, developers tend to consider the goal of such clone detection is to help them refactor or remove duplicate code. With that in mind, and when the judgement of a potential Type-3 clone can be subjective, industry developers tend to judge by whether it can be refactored, thus much less Type-3 clones. This results directly in more False Positives and less False Negatives for the clone detection algorithms, therefore poorer precision and better recall observed in the industrial project.

**Summary**. The distributions of different types of clones are different for the labelled samples from open source projects and industrial project. Industrial developers tend to take refactorability into account for the judgement of potential Type-3 clones. Therefore, clone detection algorithms should take such factors into account to be more practical in industrial use case.

## V. Threats to Validity

**Internal Validity**. The authors of SourcererCC used 6 lines/50 tokens as the minimum size of methods to be considered, whereas we used 10 lines in our experiments. We chose 10 lines because it is a common cutoff used in the majority of clone detection research, including the AutoenCODE paper. Furthermore, smaller methods are likely to include more uninteresting or trivial clones that are easier to detect [2]. Therefore, if we were to include methods with 6-10 lines, the performance of SourcererCC measured in the context of our experiments is likely to be slightly better than that reported in this paper, thus it would still be comparable with the results from the original authors.

**External Validity**. The performance of the algorithms were evaluated on 20 open source Scala projects from *Github*, it may not be the same beyond these 20 projects. We mitigate this threat by selecting the most popular Scala projects on *Github*, measured by number of stars. These projects are more likely to be forked or referenced by Scala program developers, such that the clone patterns may be carried on to many other Scala projects. Therefore, the performance measured on these 20 projects should be representative for that on most Scala projects.

The performance differences seen in the industrial project studied in this paper may not generalise to other industrial projects. However, this industrial project was developed in an environment where access to external code sources was very limited, which is a common practice in many companies in order to protect their proprietary source code. Code created in such an environment is likely to have a similar cloning pattern. Therefore, the evaluation on the industrial project in this paper should be representative for other industrial projects, and the threat is thus mitigated.

## VI. Related Work

Code clone detection is widely studied. In this paper, we aim at studying the code clones in Scala programs and focusing on the difference from existing empirical results. We list related work as follows.

Sheneamer et al. [4] surveyed different types of clone detection techniques up to 2016. They categorised clone detection techniques into: textual approaches, lexical approaches, syntactical approaches, and semantic approaches, where the approaches in each category are more complicated than the category immediately preceding it. The performance of the algorithms were extracted from other references, whereby different datasets were used, but none was based on a private and closed codebase. Sajnani et al. [8] proposed SourcererCC, a token-based clone detection algorithm. Despite the principle of the algorithm being simple, the authors applied several optimisations to eliminate impossible clone pairs as early as possible, and used an inverted index to make the algorithm scalable to very large codebases. White et al. [17] and Tufano et al. [18] proposed and improved a Deep Learning-based clone detection algorithm that automatically learned features from four different representation of the source code at different levels of abstraction. The proposed algorithm identified potential code clones by calculating the similarities between code segments using those learned features. Saini et al. [22] used a pipeline framework to identify code clones, where each stage of the pipeline used varied metrics of the code segments. Their results revealed that the framework had better detection of Type-3 and Type-4 clones, which were generally difficult to detect for other clone detection algorithms. Despite the aforementioned researchers having evaluated different clone detection algorithms on datasets as large as 100 millions of lines of code, none have yet to evaluate the performance on private industrial codebase.

Bellon et al. [24] and Roy et al. [25] compared the performance of different clone detection algorithms and tools up to 2007 and 2009. To make an unbiased comparison, the algorithms and tools were applied on the same datasets, which were relatively small (less than one million lines of code). Svajlenko et al. [26] extended the previous comparison framework for clone detection algorithms and tools, and introduced their mutation and injection framework. Ragkhitwetsagul et al. [16] compared clone detection techniques as well as plagiarism detection and compression tools up to 2018, in a scenario by scenario basis. When different algorithms and tools were compared with each other over a common dataset, it posed some requirements on the dataset, including programming language and size of the dataset. As a result, all of the tool comparison works were done over some open source codebases. Svajlenko et al. [27] introduced **BigCloneEval**, a clone detection tool evaluation framework, which makes it easier for new clone detection algorithms to be evaluated over a common open source dataset. Regarding functional languages, Xu et al. [28] investigated how developers used the unique features of Scala language.

There are also studies regarding code clones on industrial codebases. Monden et al. [29] looked at the correlation between code clones and software quality in a quantitative way. They used a token-based algorithm to identify code clones in an industrial codebase consisting of more than one million lines of code, but the performance of the algorithm was not evaluated, as it was not the focus of the paper. Zhang et al. [30] investigated the reasons behind code cloning from the perspective of developers and organisations, by analysing code

clones and interviewing developers. They also applied a token-based algorithm to detect code clones in a large industrial codebase consisting of more than 14 millions lines of code. A number of sampled code clones were manually inspected, but the performance of the algorithm was not reported, as it was not the focus of their paper.

## VII. Conclusion

In this paper, we revisited the performance of two state-of-the-art clone detection algorithms, SourcererCC and AutoenCODE, in their adaptation to Scala. Our experiments are conducted on an industrial project and 20 open source projects. We found that both SourcererCC and AutoenCODE retained consistent performance on open source Scala projects, when precision is concerned. However, we observed that the precision dropped substantially when they are applied on the industrial project.

Further investigation shows that there are less Type-2 clones in the industrial project, which could be caused by the limited access to other code sources, and there are substantially less Type-3 clones in the industrial project. Our initial discussion with industrial developers suggests that they consider unrefactorable clones unfavorable in the clone detection results, and they tend to classify such code segments as non-clones. Such observation motivates further investigation on the cloning practices and adaptation of clone detection algorithms in industrial use cases.

In the future, we would like to extend this study to investigate in details what are considered clones from industrial point of view, and how we could adapt such view to guide Clone Detection algorithms to be more practical in industrial cases. Furthermore, we want to demonstrate that Clone Detection techniques can be used in various software optimisation studies [31]–[33], as additional restriction or even optimisation objective during the optimisation process.

## Acknowledgement

The work is supported by the National Key R&D Program of China under Grant No. 2018YFB1003901, and the National Natural Science Foundation of China under Grant No. 61872273.

## References

[1] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *14th IEEE MSR'17*, 2017, pp. 291–301.

[2] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *18th Symposium on Foundations of Software Engineering*, 2010, pp. 147–156.

[3] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, pp. 1165 – 1199, 2013.

[4] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *IJCA*, vol. 137, no. 10, pp. 1–21, 2016.

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st ICSE*, 2009, pp. 485–495.

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

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

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *38th ICSE*, 2016, pp. 1157–1168.

[9] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Static Analysis Symposium*, 2001, pp. 40–56.

[10] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *2006 13th IEEE WCRE*, 2006, pp. 253–262.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th ICSE*, 2007, pp. 96–105.

[12] I. Keivanloo, C. K. Roy, and J. Rilling, "Sebyte: A semantic clone detection tool for intermediate languages," in *ICPC*, 2012, pp. 247–249.

[13] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE TSE*, 2018.

[14] M. Funaro, D. Braga, A. Campi, and C. Ghezzi, "A hybrid approach (syntactic and textual) to clone detection," in *IWSC*, 2010, pp. 79–80.

[15] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *IEEE ICSM*, 2010, pp. 1–9.

[16] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, pp. 2464–2519, 2018.

[17] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *31st ASE*, 2016, pp. 87–98.

[18] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th MSR*, 2018, pp. 542–553.

[19] R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, and C. D. Manning, "Semi-supervised recursive autoencoders for predicting sentiment distributions," in *2011 EMNLP*, 2011, pp. 151–161.

[20] A. Bosu, J. C. Carver, C. Bird, J. D. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE TSE*, pp. 56–75, 2017.

[21] M. Salam and S. U. Khan, "Challenges in the development of green and sustainable software for software multisourcing vendors: Findings from a systematic literature review and industrial survey," *JSEP*, vol. 30, no. 8, 2018.

[22] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *FSE*, 2018, pp. 354–365.

[23] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE ICSME*, 2015, pp. 131–140.

[24] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE TSE*, vol. 33, pp. 577–591, 2007.

[25] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[26] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *2014 IEEE ICSME*, 2014, pp. 321–330.

[27] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *2016 IEEE ICSME*, 2016, pp. 596–600.

[28] Y. Xu, F. Wu, X. Jia, L. Li, and J. Xuan, "Mining the use of higher-order functions: An exploratory study on scala programs," in *Proceedings of the National Software Application Conference of China (NASAC 2019). to appear*, 2019.

[29] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *8th IEEE Symposium on Software Metrics*, 2002, pp. 87–94.

[30] G. Zhang, X. Peng, Z. Xing, and W. Zhao, "Cloning practices: Why developers clone and what can be changed," in *2012 28th IEEE ICSM*, 2012, pp. 285–294.

[31] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, "Darwinian data structure selection," in *ESEC/FSE 2018*. New York, USA: ACM, 2018, p. 118–128.

[32] M. Basios, L. Li, F. Wu, L. Kanthan, and E. Barr, "Optimising darwinian data structures on google guava," in *Search Based Software Engineering*. Springer International Publishing, 2017, pp. 161–167.

[33] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *GECCO 2015*. New York, USA: ACM, 2015, p. 1375–1382.