Empirical Software Engineering (2019) 24:2236–2284
https://doi.org/10.1007/s10664-019-09697-7

Siamese: scalable and incremental code clone search via multiple code representations

Chaiyong Ragkhitwetsagul[1] · Jens Krinke[2]

Published online: 5 March 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
This paper presents a novel code clone search technique that is accurate, incremental, and scalable to hundreds of million lines of code. Our technique incorporates multiple code representations (i.e., a technique to transform code into various representations to capture different types of clones), query reduction (i.e., a technique to select clone search keywords based on their uniqueness), and a customised ranking function (i.e., a technique to allow a specific clone type to be ranked on top of the search results) to improve clone search performance. We implemented the technique in a clone search tool, called Siamese, and evaluated its search accuracy and scalability on three established clone data sets. Siamese offers the highest mean average precision of 95% and 99% on two clone benchmarks compared to seven state-of-the-art clone detection tools, and reported the largest number of Type-3 clones compared to three other code search engines. Siamese is scalable and can return cloned code snippets within 8 seconds for a code corpus of 365 million lines of code. Using an index of 130,719 GitHub projects, we demonstrate that Siamese's incremental indexing capability dramatically decreases the index preparation time for large-scale data sets with multiple releases of software projects. The paper discusses the applications of Siamese to facilitate software development and research with two use cases including online code clone detection and clone search with automated license analysis.

Keywords Code clone search · Code search engine

1 Introduction

Code search is becoming increasingly important when considering the plethora of source code currently proliferating on the Internet (Sadowski et al. 2015). Developers prefer to

Communicated by: Yasutaka Kamei

✉ Chaiyong Ragkhitwetsagul
chaiyong.rag@mahidol.ac.th; cragkhit@gmail.com

[1] Faculty of Information and Communication Technology, Mahidol University, 999 Phuttamonthon 4 Road, Nakhon Pathom, Salaya 73170, Thailand

[2] Centre for Research on Evolution, Search and Testing, Department of Computer Science, University College London, Gower Street, London, WC1E 6BT, UK

reuse coding solutions from online sources, such as Stack Overflow, instead of official documentation or books (Acar et al. 2016). Researchers have also leveraged large amounts of online code snippets to make suggestions to developers during development (Keivanloo et al. 2014; Park et al. 2014; Ponzanelli et al. 2013, 2014). Online code snippets may be exploited for program repair (Ke et al. 2015) or code examples (Keivanloo et al. 2014; Nasehi et al. 2012). On the other hand, reusing code from online sources have been found to introduce negative effects to software quality (Abdalkareem et al. 2017; Acar et al. 2016) or to violate software licenses (An et al. 2017; Baltes and Diehl 2018).

Text search engines (e.g., Google) are not designed for source code, hence they can only locate exactly similar code snippets (a challenge for clone recall) or ignore the source code's structure (a challenge for clone precision). Current code clone detection tools and techniques offer high clone detection accuracy but are not effective at finding clone candidates in a very short time (a scalability challenge). To locate or study clones from online sources as previously mentioned, a special type of code search, namely code clone search, which accepts a code fragment as a query and performs a code-to-code search in large code corpora (Kim et al. 2018; Nishi and Damevski 2018) is needed.

To tackle the scalability and clone recall and precision challenges in large-scale clone search, this paper presents and evaluates a scalable code clone search technique that retrieves Type-1 to Type-3 code clones in seconds, and supports incremental changes in software projects. The novelty of the paper lies in using multiple code representations (i.e., multi-representation) to locate clones. Unlike other clone detection or clone search tools that are based on a single representation of code (e.g., a sequence of string, a sequence of tokens, or a set of tokens), we employ four code representations to locate clones. Moreover, we mine token frequencies in a code corpus on-the-fly and automatically adjust a query's length to improve the search speed and accuracy. Our technique also allows incremental updates to the source code bases being analysed. We implemented a tool for the technique, called **Siamese** (**S**calable, **i**ncremental, **a**nd **m**ulti-repr**ese**ntation) clone search engine. The evaluation of Siamese shows that the technique scales to 365 MLOC and returns the results within 8 seconds. Our technique offers a search precision of 95% and 99% on two established clone benchmarks, which are higher than seven state-of-the-art clone detection tools. The technique reports the largest number of Type-3 clones compared to three code search engines. Moreover, the technique also exhibits high recall and precision for all clone types in the BigCloneBench (Svajlenko et al. 2014), a large-scale clone benchmark. This paper makes the following primary contributions.

1. A multi-representation and query reduction technique for code clone search that is accurate and scalable, and their evaluation.
2. The Siamese clone search engine[1] which is scalable and incremental, suitable for performing instant clone search on large-scale data sets, such as online code repositories.

The rest of the paper is organised as follows. Section 2 explains the background and the motivation of the paper. Section 3 describes the Siamese clone search architecture, followed by implementation details in Section 4. Experimental design is presented in Section 5 and the evaluation results are discussed in Section 6. Section 7 presents two case studies using Siamese. Threats to validity are discussed in Section 8 and related work is present in Section 9, before Section 10 concludes the paper.

[1] Tool and data sets used are available at https://github.com/UCL-CREST/Siamese.

2 Background and Motivation

It is difficult to obtain high precision, recall, and scalability at the same time in code clone search. Text-based search engines such as Bing and Google are scalable to the Internet but are not designed for source code and rely only on keyword search (Sadowski et al. 2015). Dedicated code search engines such as BlackDuck OpenHub (BlackDuck 2016), Krugle (Aragon Consulting Group 2018) or Searchcode (Boyter 2018) cannot efficiently handle code clones with modifications (Keivanloo et al. 2014; Kim et al. 2018). Hummel et al. (2010) and Koschke (2014) are among the first to propose scalable clone detection systems. However, the trade-off for the scalability is their ability to report only copy-and-paste clones or clones with variable renaming (i.e., Type-1 and Type-2 clones), while the largest number of clones found in software are clones with added or deleted statements (i.e., Type-3 clones) (Roy and Cordy 2009; Svajlenko et al. 2014). Although there are scalable clone detection and clone search techniques that can locate Type-3 clones with some level of success (Keivanloo et al. 2011a, b; Sajnani et al. 2016; Kim et al. 2018), scalably finding Type-3 clones is still an open challenge.

Retrieving a ranked list of clones is preferred over a full list of clone pairs in various contexts, such as finding similar code examples (Keivanloo et al. 2014) or searching for candidates for bug fixing (Ke et al. 2015). Code clone detectors that report a complete set of clones are not suitable for these tasks because a large number of clone pairs have to be manually investigated (An et al. 2017; Yang et al. 2017; Bauer et al. 2016). In these circumstances, the user would only need a ranked list of top *n* cloned code fragments instead (Niu et al. 2017). There have been a number of code search tools which produce a ranked list of code candidates (Grechanik et al. 2010; Inoue et al. 2012; Keivanloo et al. 2014; Kim et al. 2018; McMillan et al. 2011; Niu et al. 2017; Zhang et al. 2017). Some of the tools offer limited benefits because they rely on external code search engines which are no longer available (Inoue et al. 2012), or they need a specific intermediate data set (Kim et al. 2018) in order to work properly. To support a broad range of applications, we prefer a clone search engine that is general, works out-of-the-box and is not tied to any specific use cases or scenarios.

Moreover, to find good candidates for program repair, one looks for clones which deviate from the original buggy code (i.e., Type-3/Type-4) to increase the chance of successful repairs (Ke et al. 2015). On the other hand, to check for copy-and-paste code from online sources and investigate their license compatibility, one is interested in clones that are closer to the original (i.e., Type-1/Type-2) to reduce the manual investigation time. Thus, it is important that the clone search tool captures different types of clones and adapts the ranking to report a specific type of clones according to the users' need.

Lastly, most of the clone detectors do not handle incremental addition or deletion of software projects. Thus, adding new projects to the code base under analysis or updating existing projects would result in the need to rerun the clone detection for the complete data set. Several of the proposed techniques that support incremental clone detection do not scale to large-scale data sets (Göde and Koschke 2009; Kawaguchi et al. 2009; Nguyen et al. 2009) or do not detect Type-3 clones in sacrificing for scalability (Hummel et al. 2010; Koschke 2014).

Our technique tackles the previously mentioned challenges by incorporating a novel idea of multiple code representations and three techniques, i.e., inverted index, query reduction, and customised ranking, drawn from information retrieval.

By using multiple code representations, we capture different types of clones simultaneously without a need to change the configurations. The approach also allows customisation

of the ranking function to return a specific clone type on top of the search results. By using an inverted index structure to store and retrieve clones, we obtain the scalability for clone search. The inverted index is a widely-used data structure for a fast document retrieval. The index associates a term to a list of documents containing the term. By giving a query of search terms, only documents that match with the query terms are retrieved. It offers a short query response time and is scalable to large-scale data sets (Manning et al. 2009). It has been used in scalable code clone detection tools and techniques, such as Hummel et al. (2010), Koschke (2014), Sajnani et al. (2016), Kim et al. (2018), and Saini et al. (2018), and has shown to offer high scalability on large code corpora.

3 Siamese Clone Search Architecture

We designed the architecture of our clone search approach by adopting inverted index and code clone detection techniques as depicted in Fig. 1. Source code from code corpora is stored in an inverted index, which is a widely-used data structure for fast querying of relevant documents (Manning et al. 2009). Our architecture separates the necessary indexing of source code, where the search index is created, from querying, where the clones of a queried code fragment are retrieved. Inverted index and tf-idf-based scoring functions are exploited as the infrastructure of code retrieval and similarity measurement. Siamese works at token level which supports scalable detection of near-missed clones (Kim et al. 2018; Sajnani et al. 2016). The two techniques normally found in token-based clone detection including token normalisation (Kamiya et al. 2002; Prechelt et al. 2002; Roy and Cordy 2008) and *n*-gram generation (Burrows et al. 2007; Ohmann and Rahal 2014; Prechelt et al. 2002; Schleimer et al. 2003) are performed during indexing and querying time.

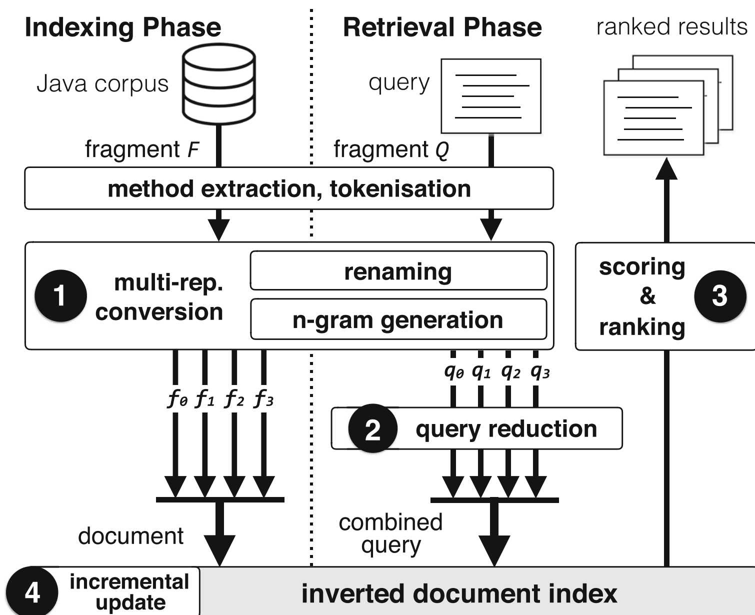


Fig. 1 Siamese architecture

The architecture incorporates a novel multi-representation and query reduction technique to increase clone search precision and flexibility of clone matching. The **multi-representation module** (Section 3.3) enables clone detection based on multiple code representations instead of one representation as in other tools. The **query reduction module** (Section 3.4) leverages the knowledge of token document frequency in a code corpus to improve the quality of the query on-the-fly. Our **customised scoring and ranking module** (Section 3.5) computes scores for matched code fragments and returns a ranked list of clones. Lastly, the **incremental update module** (Section 3.6) allows the user to add new code fragments to the index or delete selected existing code fragments from the index without affecting other indexed code fragments. Siamese performs a two-phase approach: an indexing and a querying phase.

3.1 Indexing Phase

In this phase, Siamese processes a given source code base(s) to generate a searchable code index. Siamese supports two types of code fragments, files and methods, and the input code fragments are preprocessed before being stored into the inverted index. Siamese is a token-based tool and is resilient to incomplete or uncompilable code fragments. If the method parsing fails, it falls back to store the source code at a file level. Each input code fragment F (file or method depending on its granularity) is then tokenised into a stream of tokens and sent to the *multi-representation (MR)* conversion module to generate four code representations which capture the code structure at different levels, before being stored in the index. Indexing source code files is an expensive task because the tool has to process all the available code data. Fortunately, it occurs far fewer times than the querying phase.

3.2 Querying Phase

The querying phase happens when the clone search tool receives a code query from its user and returns clone results. Only indexed documents containing the query terms are retrieved and ranked. Querying is the main activity for Siamese and usually occurs many more times than indexing. In this phase, the source code query is prepared in the same way as in the indexing phase by passing through method extraction and tokenisation steps. The query phase also supports two types of code fragments: files and methods. A tokenised code query Q is sent to the MR module to generate four query representations, i.e., siblings. The *query reduction (QR)* module rewrites and generates reduced queries from the original four query siblings. The reduced query siblings are combined into a single search request and executed on the search engine. Siamese retrieves indexed code fragments that match with the combined query and computes the ranking of results using a customised scoring function before reporting them to the user.

3.3 Multi-Representation (MR)

The Siamese clone search approach works with four code siblings derived from the original source code fragment F by the multi-representation module. To be able to detect clones of each clone type with a high precision, each clone type has a dedicated code representation. The set of four code representations $\{r_0, r_1, r_2, r_3\}$ that represent F are defined as follows.

1. **Original representation** r_0: A stream of tokens, i.e., 1-grams, containing tokens from the original source code (text search).

Table 1 Representative tokens for specific token types

D	data types	J	Java class names
K	Java keywords	P	Java packages
O	operators	S	string literals
V	numbers	W	words (other identifiers)

2. **Type-1 representation** r_1: A stream of n-grams containing tokens from the original code (Type-1 clone search).
3. **Type-2 representation** r_2: A stream of n-grams containing normalised n-grams with identifier, literal, and type tokens replaced by the representative tokens (Type-2 clone search).
4. **Type-3 representation** r_3: A stream of n-grams containing normalised n-grams with all tokens replaced by the representative tokens, except Java punctuators {, }, `[`, `]`, `(`, `)`, and `;`. Punctuators are not normalised as they are meaningful to the code structure (Type-3 clone search).

The three n-gram-based representations (r_1, r_2, r_3) are derived from the stream of tokens in the original representation (r_0). Our MR module augments the normal text search and makes it more suitable for code search by including three more representations that leverage token types, code structure, and the knowledge of clone types. For Type-1 representation (r_1), the n-grams are generated directly from the original representation (r_0). For Type-2 (r_2) and Type-3 representation (r_3), the stream of tokens r_0 is normalised to a reduced token stream in which tokens of specific types are replaced by a representative token. Table 1 shows the list of our pre-defined representative tokens containing D for data types, J for Java class names, K for Java keywords, P for Java packages, O for operators, S for string literals, V for numbers, and W for words, i.e., other identifiers. In case of r_2, all identifiers, types, numbers, and string literals are replaced by a representative token W, D, V, and S respectively. For r_3, all tokens are replaced with their respective representative tokens. Then, r_1, r_2, and r_3 are obtained by n-gramising their respective reduced token stream.

For example, given a code fragment of a binary search method in Fig. 2, the four representations r_0, r_1, r_2, r_3 generated from the MR module are depicted in Table 2.

This MR technique enables Siamese to capture multiple clone types at the same time. During the search, each code representation in the query will match with its respective

```
public static int binarySearch1 (int arr[], int key,
                                 int imin, int imax) {
  if (imax < imin)
    return -1;
  int imid = (imin+imax)/2;
  if (arr[imid] > key)
    return binarySearch1(arr,key,imin,imid-1);
  else if (arr[imid] < key)
    return binarySearch1(arr,key,imid+1,imax);
  else
    return imid;
}
```

Fig. 2 An example code fragment of a binary search method

Table 2 The four representations of the `binarySearch1` method generated from the MR module

r_0 (n-gram size = 1)

```
public static int binarySearch1 ( int arr [ ]
, int key , int imin , int imax ) ... ;
else return imid ; }
```

r_1 (n-gram size = 4)

```
publicstaticintbinarySearch1 staticintbinarySearch1( intbinarySearch1(
int binarySearch1(intarr (intarr[ ...
;elsereturnimid elsereturnimid; returnimid; }
```

r_2 (n-gram size = 4)

```
publicstaticDW staticDW( DW(D W(DW (DW[ DW[] W[],
[],D ],DW ,DW, DW,D W,DW ,DW) DW) W){if ){if( ...
);elsereturn ;elsereturnW elsereturnW; returnW; }
```

r_3 (n-gram size = 4)

```
KKDW KDW( DW(D W(DW (DW[ DW[] W[], [],D ],DW ,DW,
DW,D W,DW ,DW) DW){ W){K ){K( K(W K(WO (WOW ...
KK(W WOV, OV,W V,W) ,W); W);K ;KKW KKW; KW; }
```

representation of the indexed code fragments. We apply MR conversion to the source code in both the indexing and querying phase. In the indexing phase, Siamese creates a new document for a given code fragment and puts the four representations in separated fields inside the document. Then, the document is stored in the search index. In the querying phase, Siamese creates a combined query containing four sub queries of the four representations.

3.4 Query Reduction (QR)

Clone search suffers the long query problem (Kumaran and Carvalho 2009) since a code fragment is given as a query. To tackle this problem, we adopted a query reduction technique using token document frequency (*df*), i.e., the number of documents in which the token appears, as a query quality predictor (Kumaran and Carvalho 2009). We rewrite the query to contain only rare tokens and discard frequent ones. According to studies of Zipf's power law in software (Zipf 1932; Knuth 1971; Zhang 2008), there are a few highly frequent tokens in programming languages and the frequency of tokens drop rapidly inversely proportional to their ranks. Thus rare code tokens are ranked among the last and share only a few documents with others. By choosing only rare tokens to form a reduced query, one can (1) decrease the number of retrieved code snippets to be only highly relevant ones, (2) increase the search speed due to fewer search terms to process, and (3) avoid false positive results. Our query reduction technique chooses rare tokens in a query on-the-fly by analysing *df* scores of all query tokens.

Siamese derives four sibling queries q_0, q_1, q_2, q_3 from the original query Q, i.e., a given code fragment, and shortens all of them. The QR module gets rid of duplicated tokens by consolidating tokens or n-grams in q_0, q_1, q_2, q_3 into a set of unique tokens and n-grams. Then it filters the tokens based on their *df* score. For each representation q_i, tokens with *df* score lower than or equal to a threshold θ_i are kept in the reduced query, and tokens with *df* score higher than θ_i are discarded. The θ_i value is a proportion of the number of documents

in the index and can be adjusted via a variable called $dfCap_i$ (ranging from 0 to 100 percent). The threshold θ_i and each reduced token query q'_0, q'_1, q'_2, and q'_3 are defined as below.

$$\theta_i = \text{dfCap}_i \times |\text{documents}|, i \in [0, 3]$$
$$q'_i = \{t \in q_i : df(t) \leq \theta_i\}, i \in [0, 3]$$

The optimal θ_i value for the four representations may be different based on the distribution of tokens and *n*-grams in each representation. Setting a low θ value offers high discriminative power since it allows only rare tokens to appear in the query, and results in a short query, while selecting a high θ_i value gives low discriminative power and allows frequent tokens to be included in the query.

3.5 Scoring and Ranking of the Results

Siamese exploits Apache Lucene's scoring and ranking function to create a list of ranked cloned results. The scoring and ranking technique is based on a *vector space model (VSM)* (Salton et al. 1975) representation by converting documents, i.e., code fragments, into k-dimensional weight vectors $V = \langle w_1, w_2, w_3, ..., w_i, ..., w_k \rangle$ where k equals the number of terms in the dictionary. A popular weighting scheme is *term frequency (tf)* and *inverse document frequency (idf)*. *tf* represents how frequent a term occurs in a document and is defined as $\text{tf}(t, d) = \sqrt{\text{frequency}(t, d)}$. *idf* represents how often the term occurs across all the documents in the corpus and is defined as $\text{idf}(t) = 1 + \log(\frac{|\text{documents}|}{df(t)+1})$, where $df(t)$ stands for document frequency of term t.

Apache Lucene computes a *relevance score* between a query vector and a document vector in order to gain speed in searching and ranking. Relevant documents are ranked according to their scores, i.e., their relevancy to the query, before returning to the user. The Lucene scoring formula (Elasticsearch 2012) is

$$\text{score}(q, d) = \sum_{t \in q} [\text{tf}(t, d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t, d)] \cdot \text{queryNorm}(q) \cdot \text{coord}(q, d), \quad (1)$$

where a $\text{score}(q, d)$ between a document d in the index and the query q is computed from a sum of term scores for all the terms in q. A score for each term t in the query is computed from a multiplication of the term frequency in document $\text{tf}(t, d)$, the squared inverse document frequencies $\text{idf}(t)^2$, the term boosting weight $t.\text{getBoost}()$, and the field length normalisation $\text{norm}(t, d)$. Finally, the sum of term scores is multiplied by a query normalisation factor, $\text{queryNorm}(q)$, and a query coordination, $\text{coord}(q, d)$.[2]

Since $\text{tf}(t, d)$ will be zero for terms that do not exist in the document, only matched terms contribute to the score. Siamese relies on four representations of Java code, hence the final score of each code snippet is a sum of scores from the four reduced queries q'_0, q'_1, q'_2, and q'_3. Our customised scoring function is

$$\text{score}_{\text{Siamese}}(Q, d) = \sum_{i=0}^{3} \text{score}(q'_i, d). \quad (2)$$

[2]Detailed explanation: a query normalisation factor, $\text{queryNorm}(q)$, enables a comparison between results of different queries; query coordination, $\text{coord}(q, d)$, gives higher scores to documents that contain a high percentage of terms in the query; query boosting, $t.\text{getBoost}()$, gives a boosted term more importance than another; and field length normalisation, $\text{norm}(t, d)$, gives higher weight to a shorter field than a long field in case a document is represented by more than one field, e.g., title and body.

In addition, during computation of the reduced query scores, we assign a specific query term boosting weight for each representation, t.getBoost(), equals the size of n-gram. The terms in q'_0 are not boosted, i.e., t.getBoost() $= 1$, since the original code tokens are 1-gram and can match relatively more frequently compared to other representations (we empirically validated this in Section 4.2). In contrast, the search terms in the n-gram-based representation q'_1, q'_2, q'_3 receive a higher query boosting weight. For example, if we choose the n-gram size for the query terms in q'_1 at 4, the matched n-grams in q'_1 will receive a boosting weight of 4. Since the query boosting score equals to the size of n-gram, the larger chosen n-gram size for each representation, the higher the query boosting weight is given and the higher score is received when terms in that representation find a match. We later explore that this boosting scores can be adjusted to accurately search for a specific clone type.

Finally, after the scores have been computed, the candidates are ranked based on their scores from the highest to the lowest. If two documents obtain the same score, they are sorted based on the alphabetical order of the file and method names. Siamese then returns the top n results from the ranked list to the user.

3.6 Incremental Updates

Siamese allows incremental updates to its index which is beneficial for maintaining an index of large-scale code repositories where the index can be updated to new changes without a need to reindex all the repositories again, similar to Hummel's work (Hummel et al. 2010). With large-scale source code data, it becomes a necessity for code clone detection or clone search tool to handle changes in code bases incrementally. Siamese leverages the flexibility of inverted index to allow its user to add, edit, delete code fragments in the index without affecting other indexed code fragments. For addition, the user can tell Siamese to incrementally add a given code fragment or project(s) to its index instead of recreating the index from scratch. For deletion, Siamese uses a given wildcard pattern for matching with the project or file name of code fragment(s) intended to be deleted and performs a deletion on the matched fragments. An update operation can be done using a deletion followed by an addition.

4 Siamese Implementation

Our implementation of Siamese utilises *Elasticsearch* (Elasticsearch 2016), an open-source high performance distributed full-text search engine, for a scalable code indexing and retrieval. The current implementation is in Java and uses a single Elasticsearch node with one shard. We built the preprocessing, MR module, QR module, and scoring function on top of Elasticsearch's infrastructure. The Java method parsing is done using the Java parser (van Bruggen 2017) and the tokenisation is done using the Antlr4 lexer with a Java 8 grammar (Parr et al. 2017). Our implementation allows the tool to be executed on a single desktop machine or in a distributed manner by increasing the number of Elasticsearch nodes.

The MR, QR, scoring and ranking modules are language agnostic while the parser, tokeniser, and normaliser are language dependent. The current implementation of Siamese supports Java. To add a new language, one has to provide an implementation of the method extractor, tokeniser, and code normaliser for the language.

4.1 Selection of *N*-Gram Sizes

The selection of the optimal n-gram size is not a trivial task. Selecting a large n-gram restricts Siamese to detect clones with small gaps of modified, inserted or deleted statements to ensure the confidence of being clones. In addition, a large n-gram size encodes more information in each gram and also contains a longer overlapping region between each gram, which will affect the memory required and the disk I/O time to process the n-grams. On the contrary, selecting a small n-gram allows larger gaps with better matching flexibility and requires less memory and disk access time, but also results in a higher chance of retrieving false clone pairs.

We surveyed the literature that use n-gram for clone detection and code similarity to study their choices of n-gram sizes. Burrows et al. (2007) selected 4-grams in their software plagiarism detection approach. Myles and Collberg (2005) found that the size of 4-gram or 5-gram offers a suitable tradeoff between credibility and resilience for their n-gram-based software birthmark technique. Ohmann and Rahal (2014) observed that $n = 4$ and $n = 13$ is the optimal choice for Manhattan and cosine distance respectively. We observed that 4-gram was chosen and shown a good performance in the three studies. Thus, we selected the n-gram size of 4 for our three code representation r_1, r_2, and r_3 in the MR module. 4-gram is long enough to capture code sequences but still allows small modifications within a statement. The representation r_0 relies on 1-gram to function as a keyword search, which is useful when looking for a specific token among the cloned fragments.

4.2 Choosing the Query Reduction Thresholds

Similar to the n-gram sizes, the selection of appropriate query reduction thresholds is important for generating high-quality queries. We used two data sets to select the optimal threshold θ values for the QR module. First, we selected the well-known Bellon's clone benchmark (Bellon et al. 2007) for this analysis. The benchmark provides a partial clone ground truth in C and Java systems and has been used in several code clone studies (Wang et al. 2013; Svajlenko and Roy 2014; Koschke et al. 2006). The Bellon's benchmark was only used in this empirical n-gram analysis, and not used in any of Siamese's evaluation to avoid configuration bias. We used the four Java systems, java-swing (204K SLOC), eclipse-jdtcore (148K SLOC), eclipse-ant (16K SLOC), and netbeans-javadoc (19K SLOC) from the benchmark. Second, we employed the Qualitas corpus (Tempero et al. 2010). It is a curated Java corpus that has been used in several software engineering studies (Taube-Schock et al. 2011; Beckman et al. 2011; Vasilescu et al. 2011; Omar et al. 2012). The projects in the corpus represent various domains of software systems ranging from programming languages to visualisation. We selected the 20130901r release of the Qualitas corpus containing 111 Java open source projects. Since we need four threshold values for the four reduced query siblings q'_0, q'_1, q'_2, and q'_3, we derived four data sets from Bellon's benchmark and the Qualitas corpus, namely r_0, r_1, r_2, and r_3 respectively, to match with the structure of our four code representations. For r_1, r_2, and r_3, we adopted the n-gram sizes of 4 as previously discussed. Then, we counted document frequencies of the tokens and sorted them based on their frequency.

A visualisation of the term's document frequency vs. its rank from Bellon's benchmark is shown in Fig. 3. The two sub figures at the bottom show a zoomed-in version of the two original plots. We observed that the document frequency of r_0, the original tokens, dropped sharply and started rapidly converging to one at approximately 10% (2K) of all the documents in the corpus. Similar observation was found for r_1. The document frequency

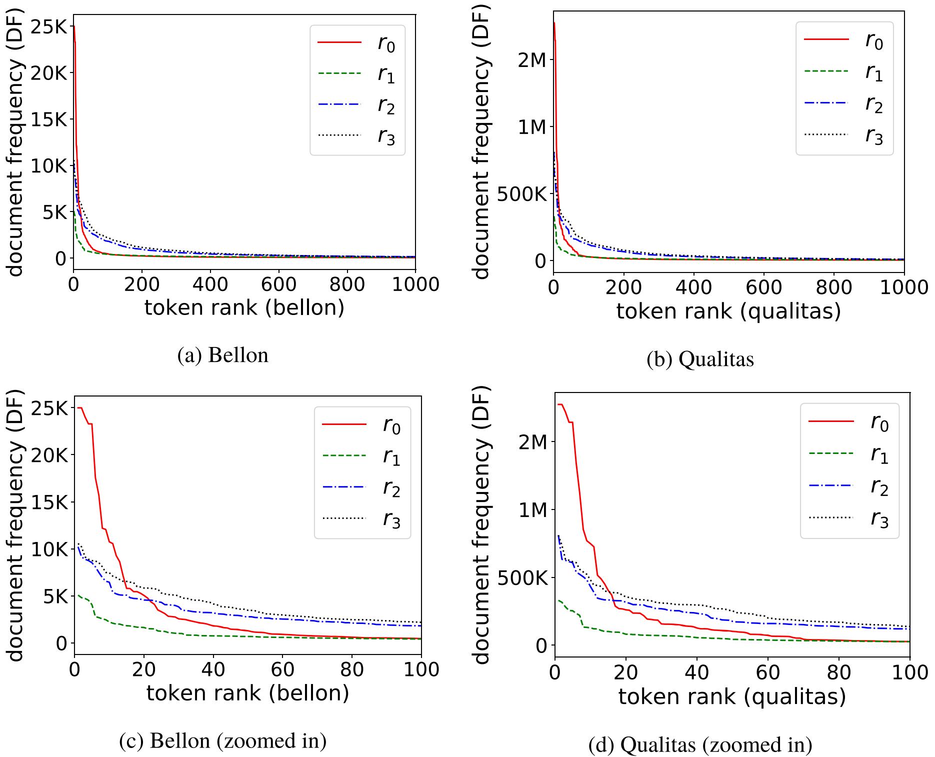


Fig. 3 Java term rank and its document frequency

of r_2 and r_3 also converged to one. They dropped to one slightly slower than r_0 and r_1 due to the token normalisation, but they were also almost distinct at 10% of all the documents. Similar findings were observed for the Qualitas corpus as depicted in Fig. 3b. With this observation, we picked the same query reduction threshold for all representations at 10%.

5 Experimental Design

We designed Siamese to be a multi-purpose clone search tool that can be exploited for various clone-related applications. To be useful, the tool must scale to the size of code corpora on the Internet while still return accurate ranked lists of clone results in a reasonable time (i.e., seconds).

We asked the following research questions to asses the practicality of Siamese to clone search applications.

RQ1: **Multi-Representation and Query Reduction:** *How effective are multi-representation and query reduction (MR-QR) to improve clone search accuracy?* To measure the effectiveness of our multi-representation and query reduction techniques, we compared the search accuracy of Siamese with MR and QR reduction to the baseline text search engine.

RQ2: Search Accuracy: *How accurate are Siamese search results?* We measured Siamese search accuracy on the three established clone data sets and and a GitHub data set. We compared Siamese to several state-of-the-art clone detection and clone search tools. The findings demonstrate the quality of Siamese's search results compared to other tools.

RQ3: Clone Ranking: *How effective is Siamese clone ranking on finding alternative implementations?* We exploited Siamese MR for a fine-grained search targeting only Type-3 clones for alternative implementations and evaluated the accuracy of the ranked results.

RQ4: Scalability: *How practical is Siamese to index and search on large-scale code corpus?* Scalability is one of the most important aspects of Siamese. We evaluated Siamese's scalability by measuring its indexing and querying time on the BigCloneBench data set containing 365M SLOC.

RQ5: Incremental Update: *How fast is Siamese's incremental update?* Using an index of 130,719 GitHub projects, we evaluate Siamese's incremental update module by measuring an index update time over hundreds of releases of the three most-starred Java software projects. The findings show the time saved by Siamese incremental index update when the user wants to update projects in the existing index.

5.1 Data Sets

We adopted three existing data sets used in other empirical code clone studies namely the OCD data set (Ragkhitwetsagul et al. 2018), the SOCO data set (Flores et al. 2014), and the BigCloneBench data set (Svajlenko et al. 2014; Svajlenko and Roy 2016) for our evaluation. Their summary is displayed in Table 3. There is a complete ground truth for the first two data sets, while there is a partial ground truth for the third data set. The OCD data set provides Java files with pervasive code modifications made by code obfuscators and compiler/decompilers. It covers clones of Type-1 to Type-4 (i.e., semantic clones or two code fragments with different syntax but share the same semantic). The OCD data set contains 100 Java files with a ground truth of 1,000 clone pairs at file-level. The 100 files consist of 10 groups of 10 files that are derived from one file and are therefore clones of each other. The SOCO data set was created for the detection of source code reuse competition. It contains clones of boiler-plate code fragments with a few or without modifications. The data set contains 259 files with a ground truth of 453 clone pairs at file-level. The OCD and the SOCO data sets were used in our previous study to compare 30 code similarity analysers (Ragkhitwetsagul et al. 2018). Third, the BigCloneBench data set is one of the largest clone benchmarks available to date. It is created from IJaDataset 2.0 (Rilling et al. 2018), a data set of 25,000 Java systems. The benchmark contains 2.9 million files with 8 million manually validated clone pairs of Type-1 up to Type-4. The BigCloneBench data set was used for clone evaluation and scalability test in several large-scale clone detection and clone search studies (Kim et al. 2018; Li et al. 2017; Sajnani et al. 2016; Svajlenko et al. 2014; Svajlenko and Roy 2015). Lastly, for the evaluation of Siamese's incremental update, we relied on a set of publicly available 130,719 GitHub Java projects.

5.2 Error Measures

We evaluated our approach to answer RQ1 and RQ2 using three error measures: precision at 10, mean average precision (MAP), and mean reciprocal rank (MRR). They are defined as follows.

Table 3 The data sets for Siamese evaluation

No.	Data set	Files	Clone pairs	SLOC
1.	OCD	100	1,000	9,618
2.	SOCO	259	453	26,122
3.	BigCloneBench	2,876,220	8,375,313	365M

Given the top n ranked results of which TP are true positives, precision at n (denoted n-prec) is defined as

$$\text{n-prec} = \frac{\text{TP}}{n}. \tag{3}$$

Precision at 10 (i.e., 10-precision) is a special case of precision at n where $n = 10$. It is used when only the top 10 results are taken into account, which reflects real-world web search scenarios that only 10 results are displayed per page (Manning et al. 2009).

Mean average precision (MAP) measures the quality of results across several recall levels where each relevant result is returned. It is calculated from multiple average precision scores (denoted APrec) obtained for the set of top k documents existing after each relevant document is retrieved, and this value is then averaged over all the queries (Manning et al. 2009). If the set of relevant documents for a query $q_j \in Q$ is $\{d_1, ...d_{m_j}\}$ and R_{jk} is the set of ranked retrieval results from the top result until we get to document d_k, then

$$\text{MAP} = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{i=1}^{m_j} \text{APrec}(R_{jk}). \tag{4}$$

Mean reciprocal rank (MRR) considers the case where only one relevant document is needed. MRR measures, on average across $|Q|$ queries, the reciprocal rank of the relevant document (i.e., clone) to each query q in the search result (Craswell 2009), i.e.,

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}. \tag{5}$$

6 Evaluation and Results

The evaluation of Siamese was performed on a single desktop computer. In RQ1, RQ2, and RQ3, Siamese was run on a MacBookPro with a single 2.9 GHz processor, 16 GB of RAM, and 512 GB of SSD. In RQ4, Siamese was run on a CentOS 7.4.1708 machine with eight 3.00 GHz processors, 32 GB of RAM, and 500 GB SATA disk. In RQ5, Siamese was run on an Ubuntu 16.04.4 LTS machine with eight 3.70 GHz processors, 32 GB of RAM, and 2.8 TB SATA disk.

6.1 RQ1: Multi-Representation and Query Reduction

How effective are multi-representation and query reduction (MR-QR) to improve clone search accuracy?

Table 4 Search performance (MAP) with different combinations of code representations

(a) One and two representations

Data set	1 rep.				2 reps.					
	r_0	r_1	r_2	r_3	r_{01}	r_{02}	r_{03}	r_{12}	r_{13}	r_{23}
OCD	0.785[a]	0.921	0.889	0.892	0.850	0.844	0.842	0.923	**0.938**	0.900
SOCO	0.977[a]	**0.990**	0.948	0.939	0.987	0.964	0.960	0.979	0.978	0.942

(b) Three and four representations

Data set	3 reps.				4 reps.
	r_{012}	r_{013}	r_{023}	r_{123}	r_{0123}
OCD	0.885	0.882	0.865	0.930	0.900
SOCO	0.979	0.978	0.956	0.971	0.976

[a]using the same representation as the baseline (token-based keyword search)

To answer RQ1, we used the two data sets for which we knew the complete ground truth and measured the improvement of clone search offered by the multi-representation and query reduction (MR-QR) technique using MAP. To observe the clone search improvement offered by the MR module; the QR module; and the combination of MR-QR to a traditional search engine, we compared the baseline text search engine represented by Elasticsearch to three variants of Siamese: (1) Siamese with MR, (2) Siamese with QR, and (3) Siamese with MR-QR. The baseline represents code search engines that rely on keyword search of source code fragments, and do not take code structure into account. Moreover, the baseline of Elasticsearch text search engine is adopted by GitHub to search for code in its 8 million code repositories.[3] Thus, the baseline also represents the code search capabilities of GitHub. For the OCD data set, we retrieved 100 queries from the ground truth and expected 10 clones at the top for each query result ($r = 10$). For the SOCO data set, the 453 clone pairs in the ground truth came from 115 unique files, which we used as the queries. The number of relevant results r for each query was varied and based on the number of cloned files associated with each query as specified in the ground truth.

We started by evaluating the clone search performance based on 15 unique combinations of code representations as displayed in Table 4, denoted by the subscripted number. For example, r_{123} represents the combination of r_1, r_2, and r_3. For the OCD data set, Siamese already performed decently well using one representation especially r_1 with the MAP of 0.921. The highest MAP score was from a combination of r_{13} at 0.938. The combination of four representations (r_{0123}) received a slightly lower MAP of 0.900. However, we will show later that by using query reduction to get rid of the extraneous tokens, we could obtain even higher MAP scores than using r_1 and r_{13}. For the SOCO data set, the best combination was using a single representation of r_1 at 0.990. The r_1 representation performed well with the SOCO data set because it contained clones of boiler-plate code with very few changes (Type-1 clones). Thus, using the n-gram sequences of original tokens in r_1 would match best with the clones. The combination of four representations gave the MAP of 0.976, slightly lower than r_1.

The results from Table 4 shows that there is no single representation that performs well on both data sets. We could sacrifice some level of search precision by combining

[3]https://www.elastic.co/use-cases/github

code representations to be able to locate different types of clones in different data sets without changing the configurations. This supports our intuition of using multiple code representation for clone search.

By adopting the multi-representation alone, we could gain a higher MAP score than the baseline by up to 15% (from 0.785 to 0.900). By applying QR on top of the baseline text search, we also received an improvement. As displayed in Table 5, The MAP score on OCD increased by about 18% (from 0.785 to 0.926). However, we observed a slight decrease of MAP after applying QR for the SOCO data set with the MAP score decreased from 0.977 to 0.975. Thus, using QR alone can be beneficial only in some cases. Nevertheless, after combining MR and QR together, we always obtained the highest MAP for both two data sets. The MAP scores increased to 0.953 (21% improvement) for OCD and to 0.991 (1.4% improvement) for SOCO. We can see that the strength of MR comes with a combination with QR, and it would be best to have both modules working at the same time. However, storing four code representations in a code search index occupies roughly four times of disk space of only having a single code representation. Users of Siamese can also choose to turn off MR when storage space is limited, and only enjoy the benefits of QR.

To confirm our findings of improvements by MR-QR, we performed a statistical test using a two-tailed non-parametric randomisation test due to its robustness in information retrieval (Smucker et al. 2007). Our null hypothesis (H_0) was that there is no significant difference between the results from the baseline to the results of Siamese using MR-QR. We performed the test using 100,000 random samples with a confidence interval value of 99% (i.e., $\alpha \leq 0.01$). The values in bold represent a statistically significant improvement which rejects the null hypothesis. We found that MR-QR helps to improve the clone search precision with statistical significance compared to the baseline on the OCD data set. The improvement for SOCO was not statistically significant due to an already high MAP score reported by the baseline. We complemented the statistical test by employing a non-parametric effect size measure called Vargha and Delaney's A_{12} measure (Vargha and Delaney 2000) to measure the level of differences between two populations and found that the effect size on the OCD data set is large (0.743), while on the SOCO data set is negligible (0.522). These findings show that MR-QR is highly effective against clones with several modifications applied to the source code, and mildly effective against clones with boiler-plate code or exact code copies.

To answer RQ1, the adoption of MR-QR improves the clone search performance compared to the baseline text search engine with statistical significance. The inclusion of MR and QR alone increases the search accuracy in most of the cases.

6.2 RQ2: Search Accuracy

How accurate are Siamese search results?

Table 5 Search performance improvement (MAP) after adding multi-representation and query reduction

Data	Settings				p-value	A_{12}
	Baseline	MR	QR	MR-QR		
OCD	0.785	0.900	0.926	0.953	0.000	0.743
SOCO	0.977	0.976	0.975	0.991	0.205	0.522

We utilised all three clone data sets and a GitHub data set to measure Siamese's search accuracy (recall and precision). Each of them is discussed separately below.

6.2.1 OCD and SOCO

For the OCD and SOCO data set, we compared Siamese using MAP to seven state-of-the-art clone detectors at file level. The other clone detectors included SourcererCC (Sajnani et al. 2016), CCFinderX (Kamiya et al. 2002), Deckard (Jiang et al. 2007), iClones (Göde and Koschke 2009), JPlag (Prechelt et al. 2002), NiCad (Roy and Cordy 2008), and Simian (Harris 2015). For CCFinderX, Deckard, iClones, JPlag, NiCad, and Simian, we relied on the results reported by our previous study (Ragkhitwetsagul et al. 2018) based on the same data sets. For SourcererCC, we followed the method shown by the same study to automatically compute a similarity score based on the clone pairs reported by the tools, create a ranked results based on the similarity scores, and measure MAP score. If allowed by the tools, we configured them to use method-level or file-level clone detection, rather than lines, to reduce a comparison bias to Siamese that searches for clones at file level in this experiment.

We also tuned Siamese and compared the optimised Siamese to the other tools' optimised configurations as a previous study (Wang et al. 2013) and our study (Ragkhitwetsagul et al. 2018) have shown that the default configurations of clone detectors could harm the validity of studies relying on them.

The mean average precision of Siamese compared to seven other clone detectors using their default configurations on the two data sets, OCD and SOCO, is displayed in Table 6 and Table 7. With the default configurations of n-gram sizes and query reduction thresholds (θ) derived from the empirical study, Siamese performed best on the OCD data set with

Table 6 Comparison of search performance (MAP) on the OCD data set (100 queries)

Tool	Default		Optimised	
	Settings	MAP	Settings	MAP
Siamese	r_1=4-gram, r_2=4-gram, r_3=4-gram, θ=10%, 10%, 10%, 10%	**0.953**	r_1=[4,8,12,16,20,24]-gram, r_2=24-gram, r_3=8-gram, θ=2%, 2%, 2%, 2%	0.997
Text Search	N/A	0.785	–	–
SourcererCC	similarity=80%	0.471	similarity=40%	0.848
CCFinderX	b=50, t=12	0.569	b=5, t=11	**1.000**
Deckard	mintoken=50, stride=inf similarity=1.0	0.665	mintoken=30, stride=1 similarity=0.95	0.924
iClones	minblock=20, minclone=100	0.444	minblock=10, minclone=50	0.668
JPlag	t=9	0.857	t=5	0.918
NiCad	UPI=0.30, minline=10, rename=none, abstract=none	0.457	UPI=0.50, minline=10, rename=blind, abstract=declaration	0.859
Simian	threshold=6	0.442	threshold=3, ignoreIdentifiers	0.916

Values in bold highlight the largest value in the column

Table 7 Comparison of search performance (MAP) on the SOCO data set (115 queries)

Tool	Default		Optimised	
	Settings	MAP	Settings	MAP
Siamese	r_1=4-gram, r_2=4-gram, r_3=4-gram, θ=10%, 10%, 10%, 10%	**0.991**	r_0=1-gram, r_1=[4,8,12,16,20,24]-gram, r_2=4-gram, r_3=16-gram, θ=8%, 8%, 8%, 8%	0.994
Text Search	N/A	0.977	–	–
SourcererCC	similarity=80%	0.776	similarity=60%	0.839
CCFinderX	b=50, t=12	0.942	b=5, t=9	0.982
Deckard	mintoken=50, stride=inf similarity=1.0	0.946	mintoken=30, stride=2 similarity=0.95	0.980
iClones	minblock=20, minclone=100	0.799	minblock=8, minclone=70	0.882
JPlag	t=9	**0.991**	t=8	**0.997**
NiCad	UPI=0.30, minline=10, rename=none, abstract=none	0.870	UPI=0.30, minline=5, rename=blind, abstract=literal	0.931
Simian	threshold=6	0.884	threshold=4, ignoreIdentifiers	0.978

Values in bold highlight the largest value in the column

MAP of 0.953 and for the SOCO data set, Siamese was ranked first along with JPlag with MAP of 0.991.

Regarding the optimised version, we tuned Siamese's n-gram sizes and θ to give the highest MAP score. The n-gram sizes for the three code representation r_1, r_2, and r_3 starts from 4 to 24 with an increasing step of 4 (the representation r_0 always has the n-gram size of 1). We tried the four n-gram sizes on the three representation independently and obtained 216 different combinations. The query reduction thresholds θ cover 2%, 4%, 6%, 8%, and 10% and were identically set for the four representations. Combined the two parameters together, we searched for 1,080 combinations of Siamese's configurations. The other tools' optimised configurations and their parameter search space are based on the results from our study of comparing 30 code similarity analysers (Ragkhitwetsagul et al. 2018). CCFinderX and JPlag was ranked 1$^{\text{st}}$ for OCD and SOCO with MAP of 1.000 and 0.997 respectively. Although Siamese did not give the highest MAP scores based on the optimised configurations, it still offered a very high MAP score (0.997 and 0.994) and was ranked the 2$^{\text{nd}}$ for both OCD and SOCO. Moreover, Siamese always outperformed SourcererCC, Deckard, iClones, NiCad, and Simian in both the default and the optimised configurations. Although it gave slightly lower MAP score than CCFinderX and JPlag after tuning, Siamese offered a much higher scalability than the two clone detectors as will be shown in RQ4.

The multi-representation module helped Siamese to perform well on different data sets even without tuning as we observed that the optimised MAP values were very close to the default configurations' MAP values. In practice, it is very difficult to always tune a clone

detector to their optimal performance. We could optimise the clone detectors in this study because we knew the complete clone ground truth of the OCD and the SOCO data sets as they were generated data sets. A clone ground truth does not exist in software projects. Thus, we mostly have to rely on the default configuration of the clone detection tools. Moreover, our previous study (Ragkhitwetsagul et al. 2018) shows that although we could find the tools' optimal configurations on one data set, we cannot efficiently reuse them on another data set. The results in Tables 6 and 7 suggest that Siamese's search performance, with or without tuning, was still comparable or even better than other tools with their optimised configurations. This shows that Siamese effectively handles clones with several code modifications in the OCD data set and boiler-plate code in the SOCO data set, while still offers comparable search precision to other clone detection tools optimised for the data sets.

6.2.2 BigCloneBench

The third data set is the BigCloneBench, which is a well-known data set that has been used to benchmark code clone detectors and clone search engines (Sajnani et al. 2016; Kim et al. 2018; Li et al. 2017). To be able to compare with other tools' existing results on BigCloneBench, we evaluated Siamese using recall as reported in previous work. Because Siamese is not a clone detector but a clone search tool, it does not report a set of clones that can be used to measure recall and precision. Nevertheless, we compared its performance to other tools here for a situation where it will be adapted as a clone detector.

The BigCloneBench data set's size represents code corpora on the Internet and is suitable to assess how well the tool differentiates and reports relevant code snippets from millions of candidates. Moreover, the data set offers a ground truth of 8 million clone pairs. The evaluation was performed at method level as required by the BigCloneBench oracle. We measured Siamese's clone recall by issuing multiple queries and evaluated the returned ranked results.

We followed the approach used by Kim et al. (2018), who also evaluated their clone search engine for recall, by choosing 14,780 methods that appeared in the clone oracle as the queries. Although we did not use all the methods in BigCloneBench to query, it does not affect the clone recall. The methods that do not appear in the clone oracle do not have any clone pair associated with them, thus using them to query for clones would only result in false positives, which is not taken into account for recall (on the contrast, it will affect precision). To compute the recall score, we utilised an automated tool called BigCloneEval (Svajlenko and Roy 2016) which was created for recall computation on BigCloneBench. For each query, we choose the result size of 900 to match with the settings used in the evaluation of a clone search engine, FaCoy, by Kim et al. (2018).[4] After finishing querying, we gave the result to the BigCloneEval tool for recall calculation. Tables 8, 9, and 10 shows the recall scores of Siamese on BigCloneBench compared to the other five tools including SourcererCC, CCFinderX, Deckard, iClones, and NiCad as reported in the study by Sajnani et al. (2016) and FaCoy code search tool as reported in the study by Kim et al. (2018). BigCloneBench categorised the clone pairs into Type-1 (T1), Type-2 (T2), very-strongly Type-3 (VST3) with a similarity in range of 90% (inclusive) to 100%, strongly Type-3 (ST3): 70–90%, moderately Type-3 (MT3): 50–70%, and weakly Type-3 or Type-4 (WT3/T4): 0–50% (Svajlenko and Roy 2016). Moreover, BigCloneEval divides the evaluation into 3 sets: All

[4]We could not include the FaCoy tool in our other RQs because the tool is released as a virtual machine image with an existing clone database. The only way to use the tool is via a web interface and there is no instruction on how to switch FaCoy to analyse a new data set (such as OCD, SOCO used in our study).

Table 8 BigCloneBench Recall Measurements (All Clones)

Tool	All Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4
Clone search engines						
Siamese	99	99	99	**99**	**88**	**17**
FaCoy (Kim et al. 2018)	65	90	67	69	41	10
Clone detectors (Sajnani et al. 2016)						
SourcererCC	**100**	98	93	61	5	0
CCFinderX	**100**	93	62	15	1	0
Deckard	60	58	62	31	12	1
iClones	**100**	82	82	24	0	0
NiCad	**100**	**100**	**100**	95	1	0

Values in bold highlight the largest value in the column

Clones, Intra-Project Clones, and Inter-Project Clones. We included the other tools' results for the all three sets except the FaCoy tool which reported its recall scores only for the All Clones set.

For All Clones set (Table 8), Siamese provided recall scores of 99% for T1, T2, VST3, and ST3. Siamese obtained the highest recall of 88% for MT3 compared to other tools and 17% for WT3/T4. When dividing into Intra-Project (Table 9) and Inter-Project clones (Table 10), Siamese performed slightly better on both sets with higher or the same recall scores as in the All Clones set. Interestingly, we found that Siamese could return 99% of MT3 clones in Intra-Project clones while other tools reported up to 14%. Similarly, Siamese returned 77% of WT3/T4 clones while CCFinderX and Deckard reported only 1% of the clones. A similar finding was observed for Inter-Project clones where Siamese obtained the highest recall at 87% of MT3 clones and 16% of WT3/T4 clones. The results show that the multi-representation and query reduction techniques enable Siamese to find more challenging clone pairs than state-of-the-art techniques. Although Siamese and SourcererCC share fundamental concept of index-based and token-based clone detection, Siamese can offer higher clone recall for the challenging clone types of ST3, MT3, and WT3/T4 because it

Table 9 BigCloneBench Recall Measurements (Intra-Project Clones)

Tool	Intra-Project Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4
Clone search engines						
Siamese	**100**	99	**100**	**100**	**99**	**77**
FaCoy (Kim et al. 2018)	–	–	–	–	–	–
Clone detectors (Sajnani et al. 2016)						
SourcererCC	**100**	99	99	86	14	0
CCFinderX	**100**	89	70	10	4	1
Deckard	59	60	76	31	12	1
iClones	**100**	57	84	33	2	0
NiCad	**100**	**100**	**100**	**99**	6	0

Values in bold highlight the largest value in the column

Table 10 BigCloneBench Recall Measurements (Inter-Project Clones)

Tool	Inter-Project Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4
Clone search engines						
Siamese	99	**100**	99	**99**	**87**	**16**
FaCoy (Kim et al. 2018)	–	–	–	–	–	–
Clone detectors (Sajnani et al. 2016)						
SourcererCC	**100**	97	86	48	5	0
CCFinderX	98	94	53	1	1	0
Deckard	64	58	46	30	12	1
iClones	**100**	86	78	20	0	0
NiCad	**100**	**100**	**100**	93	1	0

Values in bold highlight the largest value in the column

detects clones using four code representations (i.e., raw code tokens, Type-1 *n*-gram tokens, Type-2 *n*-gram tokens, and Type-3 *n*-gram tokens). Moreover, it does not remove any token from the code in the index. On the other hand, SourcererCC's partial indexing keeps only rare tokens of a single code representation (i.e., raw code tokens) in the clone index, which restricts the tool to find only clones that share the rare tokens. The removal of frequent tokens in Siamese occurs at query time and it only affects the tokens in the query. In addition, by using the query reduction technique, Siamese is more flexible than SourcererCC during the querying phase. It can adjust the number of rare tokens from the token frequencies in each data set. The *dfCap* parameters also allow Siamese's users to adjust the number of rare tokens in every query. In contrast, SourcererCC's optimised index keeps rare tokens based on a user's selected clone similarity threshold at the index building time, which applies to all the queries during the querying phase. However, Siamese suffers from a larger clone index than SourcererCC due to the complete collection of code tokens and also its multiple code representations.

To measure precision, there is no benchmark and standard methodology for precision measurement in clone detection and some authors relied on a manual investigation of the reported clone pairs (Sajnani et al. 2016). The BigCloneBench is created by using regular expressions derived from 43 target functionalities, i.e., Java class files, to search for clone candidates in 25,000 Java projects, followed by a manual confirmation. Thus, we chose the 43 Java files that represented the target functionalities in BigCloneBench as the search queries. We obtained 96 method queries from the 43 chosen files. The oracle of 8 million manually validated clone pairs provided by the BigCloneBench authors is only a partial ground truth as it only contains validated clone pairs but not all existing clone pairs. It is possible that Siamese reports true clones which do not exist in the ground truth during the evaluation. Thus, a manual validation is needed to obtain precision scores. To evaluate Siamese as a clone search engine that returns a ranked list of top *n* results, we relied on MRR and 10-precision for precision measurement. The two error measures are well-known in information retrieval since they reflect a real-world setting of using a search engine where only a few first results will be looked at due to a limited attention span of human investigator (Miller 1956). The first author took the role of a human investigator.

Table 11 shows the MRR and 10-precision scores based on the ground truth in the benchmark and after the manual confirmation. Siamese's search results of the 96 queries on

Table 11 BigCloneBench precision measurements

MRR	10-prec	T1	T2	T3
0.948	0.871	0	25	811

BigCloneBench offered an MRR score of 0.948 and 10-precision of 0.871. The MRR score of 0.948 shows that Siamese mostly returns true clone fragment as the first result. The 10-precision score of 0.871 shows that true clones are observed within the top ten on average 87.1% of the time. These are relatively high precision scores considering that there was no Type-1 clone for all the 96 queries and only Type-2 and Type-3 clones were available.

During our manual confirmation of the BigCloneBench clone search results, we noticed some interesting clones that were reported by Siamese. We found that in a few cases, Siamese not only reported clones that were syntactically similar to the query but also semantically similar. For example, consider the `binarySearch1` method shown before in Section 3.3 as the query. As shown in Fig. 4, the first result was very similar to the query but with differences in the data types and expressions. The second result contained a more diverse version of binary search with renamed variables and different conditional statements. Interestingly, we found that the third result is a method that performed binary search using a `while` loop instead of recursion as in the query and the fifth result was a method to search for an index number which used binary search as the underlying search algorithm.

6.2.3 False Positives

To understand the weaknesses of our approach, we summarise a few patterns found in the manually-validated false positive clone pairs from BigCloneBench. First, a number of false positive clone pairs come from a method that is declared inside another method. For example, as shown in Fig. 5, the method `deleteRecursively1` is reported as clone pairs with its three inner methods: `visitFile`, `visitFileFailed`, `postVisitDirectory`. This issue is due to the nature of the Java parser we used,[5] which, in this situation of having methods inside another method, returns both the containing and contained methods after parsing. Once both the containing and contained methods are included together during clone search, they are reported as clones because they automatically have an exact-match code region. This problem can be fixed by analysing the clone results and filtering these inner-method clone pairs out.

In addition, we observed that many of the false positive pairs are caused by two methods that share several code tokens and *n*-gram sequences. As shown in Fig. 6, the two methods perform a different task of checking a palindrome word and checking for blank string. Nonetheless, they share several similar code tokens such as `for (int i = length - 1; i >= 0; i--)`, `.charAt(i)`, or `length = original.length();`. Increasing the *n*-gram sizes may remove these false positives, while also reduce the chance of finding Type-3/Type-4 clones.

6.2.4 Comparison with Code Search Tools

Since the precision evaluations with OCD and SOCO are based on relatively small generated data sets, they may not fully represent real-world clone search scenarios where (1) each

[5] JavaParser: http://javaparser.org

```
/* 1st result (sample/BinarySearch.java, 19, 30) */
public static <T extends Comparable<T>>
    int binarySearch3(T[] arr, T key, int imin, int imax) {
  if (imax < imin) return -1;
  int imid = (imin+imax)/2;
  if (arr[imid].compareTo(key) > 0)
    return binarySearch3(arr,key,imin,imid-1);
  else if (arr[imid].compareTo(key) < 0)
    return binarySearch3(arr,key,imid+1,imax);
  else return imid;
}

/* 2nd result (default/103246.java, 20, 26) */
private int recFind(long searchKey,
                    int lowerBound, int upperBound) {
  int curIn;
  curIn = (lowerBound + upperBound) / 2;
  if (a[curIn] == searchKey) return curIn;
  else if (lowerBound > upperBound)
    return nElems;
  else {
    if (a[curIn] < searchKey)
      return recFind(searchKey, curIn + 1, upperBound);
    else
      return recFind(searchKey, lowerBound, curIn - 1);
  }
}

/* 3rd result (selected/2663331.java, 292, 299) */
double getValueForFeature(int f) {
  int imin = 0, imax = features.length;
  while (imin < imax) {
    int mid = (imin + imax) / 2;
    if (features[mid] == f) return values[mid];
    else if (features[mid] > f) imax = mid;
    else imin = mid + 1;
  }
  return 0;
}

/* 5th result (selected/541979.java, 138, 144) */
private int getIndex(int c, int start, int stop) {
  int pivot = (start + stop) / 2;
  if (c == value[pivot]) return pivot;
  if (start >= stop) return -1;
  if (c < value[pivot]) return getIndex(c, start, pivot - 1);
  return getIndex(c, pivot + 1, stop);
}
```

Fig. 4 Search results with syntactic and semantic clones

code query may or may not have clones and (2) a search is performed on a large code base. Moreover, we only compared Siamese to code clone detection tools, which are not specifically designed for code search. Thus, we replicated the experiment by Kim et al. (2018) by creating Siamese's code search index from 16,738 GitHub projects (with at least ten stars) and used the same 10 queries of the top 10 Stack Overflow posts with the highest view counts to search for clones. We configured Siamese to search for clones at a file level (i.e., search with a full code snippet) and with the maximum result size of 20 so that we can

```
public static void deleteRecursively1(Path dir) throws IOException {
  Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file,
      BasicFileAttributes attrs) throws IOException {
      Files.delete(file);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc)
    throws IOException {
      Files.delete(file);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory
                           (Path dir, IOException exc)
    throws IOException {
      if (exc == null) {
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
      } else {
        throw exc;
      }
    }
  });
}
```

Fig. 5 A false positive clone pair containing methods inside a method

measure precision at 10 and precision at 20 similar to their study. In Table 12 we compare the results for Siamese to the results reported by Kim et al. (2018) for their FaCoy tool and the two other code search tools: Searchcode (Boyter 2018) and Krugle (Aragon Consulting Group 2018). FaCoy is a clone search tool that finds semantically similar code fragments to the code query by using a query alternation approach. The tool includes additional relevant search keywords from Q&A posts (i.e., Stack Overflow) into a given search query. Searchcode and Krugle are online code search engines which support clone search. After the search using Siamese, the first author looked at the top 20 ranked clone results of each query and validated if they are true positives (i.e., clones) and classified each result into the four clone types.

The search results of Siamese, FaCoy, Searchcode, and Krugle are presented in Table 12. The numbers are the number of true positives found for each query classified into four clone types. We found that Siamese returned more true positive results than the other three tools, especially for Type-3 clones.[6] Siamese's precision at 10 and precision at 20 is 0.79 and 0.80 respectively compared to 0.58 and 0.49 reported for FaCoy (Kim et al. 2018). Thus, Siamese offers a higher search precision than FaCoy and found more clones, i.e., having a higher recall, than Searchcode and Krugle on the same set of 10 Stack Overflow queries.

[6]There were a few Stack Overflow posts which contained more than one code snippet and the table only presents the results from the largest code snippet in each post. The full results can be found from our study's website: https://github.com/UCL-CREST/Siamese

```
/* QUERY - TestPalindrome.java, 2, 13 */
public static boolean isPalindrome(String original) {
//A not very efficient example
  String reverse = "";
  int length = original.length();
  for (int i = length - 1; i >= 0; i--)
    reverse = reverse + original.charAt(i);
  if (original.equals(reverse))
    return true;
  else
    return false;
}

/* 1st RESULT - 2394080.java, 118, 125 */
public static boolean isNotBlank(String str) {
  int length;
  if ((str == null) || ((length = str.length()) == 0)) return false;
  for (int i = length - 1; i >= 0; i--) {
    if (!isWhitespace(str.charAt(i))) return true;
  }
  return false;
}
```

Fig. 6 Another false positive clone pair containing similar code tokens

However, FaCoy still reports the largest number of Type-4 clones potentially due to its query alternation approach. Nonetheless, there is a difficulty in comparing the experiment's results because the code search index differs for all four clone search tools, which is an issue that the FaCoy study also suffers from. The FaCoy authors use Searchcode and Krugle as-is with their own code search index, and use FaCoy to search on a code search index of 10,449 GitHub Java projects (which are forked at least once). We similarly used Siamese to

Table 12 Search results of the top 10 Stack Overflow Java posts with code snippets. The results of FaCoy, Searchcode, and Krugle is from Kim et al. (2018)

Query	Siamese					FaCoy				Searchcode			Krugle	
	outputs	T1	T2	T3	T4	outputs	T2	T3	T4	outputs	T1	T3	outputs	T1
Q1	20[a]	1	8	11		18		5	4					
Q2	20[a]	1		2	6	21		6	2					
Q3	20[a]			20		18		9						
Q4	20[a]			20						20	20		1	1
Q5	20	1		3		19	1	2	6	3	2	1		
Q6	20			20		9	1	3	1	20	20			
Q7	20			20		17								
Q8	20[a]		1	19	0	17	7	7						
Q9	20			7						2		2		
Q10	20			20		9		1	7					

[a]The code snippet contains multiple methods (i.e., queries). The presented values are only the queries with the largest code snippet

search on a code search index of 16,738 GitHub Java projects (which have at least 10 stars). Due to this issue, a comparison of the results of the four code search tools can only give an impression of how good the tools are on finding clones on their selected code search index.

To answer RQ2, Siamese offers the highest mean average precision on two clone benchmarks compared to seven clone detectors. Its multi-representation enables Siamese to detect challenging Type-3 and Type-4 clone pairs better than other tools, while still reserves high recall on Type-1, and Type-2 clones. It offers the highest recall scores of ST3, MT3, and WT3/T4 clone pairs on BigCloneBench. Lastly, Siamese found the largest number of Type-3 clones of the 10 highest-voted Stack Overflow code snippets compared to three other code search engines.

6.3 RQ3: Clone Ranking

How effective is Siamese clone ranking on finding alternative implementations?

In this RQ, we evaluated the Siamese clone ranking technique for retrieving alternative implementations of a code query on top of the search results. This technique is useful for various purposes such as learning from different implementations of an original piece of code, finding bug fix candidates, or a study of semantic clones. We are motivated by Ke et al. (2015)'s study in which a defective code fragment is used as a query and semantically similar clones are searched for (i.e., alternative implementations), using SMT constraints on input-output behavior. The authors implemented the technique in a tool called SearchRepair, then used the tool to search for semantically similar clones, and integrated each of the retrieved clones into the software, replacing the defective part. The results for RQ2 show that Siamese could locate the largest number of moderately Type-3 (MT3) and weakly Type-3 or Type-4 (WT3/T4) clones in BigCloneBench. Thus, in this RQ, we investigate further whether Siamese can also be used to search and rank alternative implementations on top of the retrieved clones, which then are possibly fed as repair candidates to an automated software repair technique. In this study, we define alternative implementations based on clone types. They are code fragments that perform the same task as the query (i.e., semantically similar), but are not identical (Type-1 clones) or not only differ in variable names and literal values (Type-2 clones). To put it another way, alternative implementations are Type-3 and Type-4 clones of the query. To locate such alternative implementations, the ranking of the retrieved clones is crucial since we prefer to retrieve Type-3 and Type-4 clones rather than Type-1 and Type-2 clones on top of the search results.

Although RQ2 shows that Siamese returns the largest number of Type-3 and Type-4 clones from BigCloneBench, the evaluation did not take the ranking into accounts. With the multiple code representations offered by Siamese, we are allowed to search for a specific type of clones that fits our needs. By adjusting a different boosting score for each representation at a query time, we could target clones of a specific type to be ranked on top of the search results, while discriminating against clones of unwanted types to be ranked lower.

This clone ranking is difficult or impossible to achieve by traditional clone detection tools. First, tools like CCFinderX, NiCad, or SourcererCC do not provide a ranked list of clones. So a human investigator does not know which clone pairs to start the investigation and has to rely on random sampling. Second, although we can rank the clone pairs based on their similarity score (CCFinderX and NiCad can report similarity scores), we cannot explicitly select clones of a specific type to be on top of the list. For example, let say we are searching for alternative implementations of a buggy code fragment, and we use NiCad

for this task. We do not want Type-1 or Type-2 clones because they are identical or very similar to the buggy code fragment that we currently have. Thus, we configure NiCad to find Type-3 clones. Nonetheless, since Type-3 clones subsume Type-2 and Type-1 clones by definition, we cannot use NiCad to locate *only* Type-3 clones. The Type-1 clones reported by NiCad will always have a similarity score higher than Type-2 and Type-3 clones, and will always be ranked on top. The human investigator will have to manually go through a number of Type-1 and Type-2 clones before finding the Type-3 clones that he or she is looking for. Third, most of the clone detectors locate clones based on a given similarity threshold. SourcererCC's partial indexing only keeps code tokens that form a clone pair with similarity higher than or equal to the threshold. Since this decision is made at indexing time, a change of the similarity threshold to find stricter or more relaxed clones will result in re-indexing of the code base. Other clone detectors such as Simian, Deckard, or iClones would face the same issues.

Similar to RQ2, we used the BigCloneBench index with 8.1 million code fragments and performed the clone search based on the 96 queries which represented 43 target functionalities in BigCloneBench. They were chosen again for this RQ because the 96 queries contained general functionalities that were normally found in Java programs, such as binary search, bubble sort, file copy, and extraction of a compressed file. Moreover, the benchmark's partial clone ground truth helped us in the manual clone investigation step. The maximum number of clone results to be investigated is 10.

Since the 43 target functionalities had only Type-2 and Type-3 clone pairs and did not have any Type-1 clone pair in BigCloneBench, we injected them into the index so they could appear in the search results. We intentionally added them into the search index in order to confuse the tool. Our goal was to find Type-3 clones that slightly or moderately differ from the query, so the injected Type-1 clones should not appear on top of the search results.

The adjusted boosting scores of Siamese for alternative implementation search is shown in Table 13. The original and Type-3 representations r_0 and r_3 received positive boosting scores of 10 and 1 respectively, while r_1 and r_2 received negative boosting scores of -1. This setting was suitable for finding clones that deviate from the query because the literal clones (Type-1) and parameterised clones (Type-2) were penalised with the negative boosting scores. We need to keep positive boosting scores for tokens in the original representation r_0 to get rid of false positives due to accidental structural similarity matches on r_3. We gave a higher score of ten for r_0 than one of r_3 to push Type-3 clones with similarity keywords on the top of the list. Since there was no clone detection in our study that gives ranked list of clones, we compared Siamese to the baseline text search engine, i.e., using the source code original tokens with no boosting score (boosting score equals one), and Siamese with the default configurations with the boosting scores of 1 for r_0 and 4 for r_1, r_2, and r_3.

Table 13 Type-3-only search: the boosting scores for the four code representations and the search accuracy

Tool	Representations				MRR
	r_0	r_1	r_2	r_3	
Baseline (text search)	1	–	–	–	0.4633
Siamese (default)	1	4	4	4	0.4550
Siamese (boosted)	10	−1	−1	1	0.7050

We adopted MRR to measure the search accuracy.[7] Since the goal of this RQ is to find bug fix candidates or alternative implementations, we only targeted Type-3 clones. Clones with Type-1 or Type-2 were not considered as relevant and received a zero score when computing MRR. Thus, in this case, the MRR score reflected how well the tool retrieved Type-3 clones on top of the search results, which fits perfectly well with our use case of finding alternative implementations. A user using Siamese to search for alternative implementations will try the first returned result and then try the next returned result until the result satisfies her or his requirements. If the first result matches the requirements already, he or she does not need to continue further. We consulted the BigCloneBench clone oracle to validate the returned clone pairs and their clone types. When a clone pair could not be found in the clone oracle, the first author performed a manual validation of the clones. Please note that the BigCloneBench ground truth does not contain obvious Type-4 clones. Nonetheless, some of the Type-3 clones in the ground truth, i.e., weakly Type-3 or Type-4 (WT3/T4) clones with a similarity between 0–50%, lie on a grey area between Type-3 and Type-4 clones. Thus, some of the verified Type-3 clones, which have a similarity between 0–50%, may be considered as Type-4 clones.

The MRR scores of the baseline text search and Siamese are displayed in Table 13. The baseline always returned Type-1 clone pairs on top of the search results (96 times out of 96 queries), followed by Type-2 and Type-3 clones and received an MRR score of 0.4633. The default Siamese gave a similar performance with an MRR score of 0.4550. Boosted Siamese outperformed the other two tools with a higher MRR score of 0.7050. The boosted Siamese returned Type-3 clone pairs on the top result 59 times, returned Type-1 clone pairs on the top 23 times, and did not return any correct clone pairs 14 times. Figure 7 shows an example of a query and a Type-3 clone fragment returned by Siamese. The pair are both methods to get a Fibonacci number. They share the same input/output but contain two different implementations using recursion and a for loop.

To answer RQ3, Siamese can effectively search and return a specific type of clones on top of the search results. Due to its multi-representation of code, Siamese can target which type of clones to be ranked on top while at the same time discriminates clones of unwanted types. This specific-clone-type ranking cannot be done using the existing clone detection or code search tools due to their use of a single code representation. The search is beneficial for a case where only a specific type of similar code is needed, such as finding potential bug fix candidates which are not identical to the given query.

6.4 RQ4: Scalability

How practical is Siamese to index and search on large-scale code corpus?

We evaluated Siamese's scalability by measuring the time needed to index and query various code base sizes. We created 10 sets of Java code with different sizes by randomly selecting files from BigCloneBench. The number of files in a set i, $i \in [1, 10]$, is 2^{2i}. The smallest set has 4 files (22 methods) and the largest set has 1,048,576 files (1,771,183) methods. We also added the complete BigCloneBench data set with 2.9 million files (4,870,113

[7] We were deterred from using the well-known mean Average Precision (MAP) or Normalised Discounted Cumulative Gain (NDCG) that were suitable for assessing the quality of ranked results. MAP and NDCG need a complete ground truth of relevant documents which were not the case for BigCloneBench.

```
/* QUERY - Fibonacci.java, 3, 10 */
public static int getFibonacci(int n) {
  if(n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return getFibonacci(n-1) + getFibonacci(n-2);
}

/* 1st RESULT - 2156644.java, 5, 13 */
public int getFibonacci(int n) {
  int prev[] = { 1, 1 };
  for (int i = 1; i < n; i++) {
    int aux = prev[1] + prev[0];
    prev[0] = prev[1];
    prev[1] = aux;
  }
  return prev[1];
}
```

Fig. 7 An example of Type-3/Type-4 cloned fragment returned as the 1st result

methods) as the last (11th) set. The experiment was performed on a CentOS 7.4.1708 machine with eight 3.00 GHz processors, 32 GB of RAM, and 500 GB SATA disk.

We separately measured the tools' index and query time in this RQ because we are more interested in a scenario of clone search than clone detection. In the clone detection scenario as performed by Koschke (2014) or SourcererCC (Sajnani et al. 2016), it is a one-off process. An index of code bases is created. Then, queries containing code fragments either from within the same project (intra-clone detection) or from other projects (inter-clone detection) are issued on the index to locate clones. The clone index may be kept for later uses if needed or recreated if the analysed code bases change. In this scenario, indexing and querying occur one after the other in a single execution. On the other hand, in the clone search scenario (or incremental and real-time detection as presented by Hummel et al. (2010)), an index of large code bases is persisted only once and loaded into memory whenever the clone search engine starts. The index is frequently updated to reflect the changes in the code bases. With this approach, a clone search tool allows as many queries as needed without a need to reindex the code bases again. We can tolerate slow indexing time as long as the tool offers fast querying time since it occurs much more often. Thus, measuring both the index and query time allows us to know how long it takes to prepare the index, and how long it takes to only retrieve clones.

For the indexing phase, we compared our tool to seven clone detectors: SourcererCC, CCFinderX, Deckard, iClones, JPlag, NiCad, and Simian. Since all the tools except Siamese and SourcererCC do not separate their clone detection into indexing and querying phase, we use their main command to detect clones to execute. Moreover, the other five clone detectors besides SourcererCC do not use an indexed-based approach, so we cannot directly compare their indexing time and rely on their clone detection time as the indexing time. We included them in this comparison to assess their scalability to large-scale code data. For Siamese and SourcererCC, we specifically instructed the tools to perform indexing on the given data sets. Each tool was executed using their default configurations and, if allowed by the tool, we allocated the same amount of 8 GB of memory for their execution. We measured the execution time using the `time` command. Unfortunately, during the execution

of CCFinderX, the tool reported encoding errors on several files. We needed to remove those files from the data set to run the tool, which would affect its running time. So, we decided to remove CCFinderX from this evaluation.

The tools' indexing time is displayed in Fig. 8. The plot shows the tools' execution time against the number of methods in each data set. Every tool completed their analysis of 22, 50, 178, 423, 1723, 6.6K, and 28K methods with increasing execution time. Deckard reported clones in the 28K set in 7 hours 14 minutes and did not return any result on the 111K set within a week, so we decided to terminate the tool's execution. iClones and JPlag finished their executions on the 6.6K set in 3 minutes and 15 minutes respectively and ran out of memory on the 28K set. NiCad threw an error in cross-clone analysis on the 442K-method set. Simian reported clones in the 28K set within 1 hour and 48 minutes and failed to analyse the 442K set.

Siamese and SourcererCC were the only two tools that could complete their indexing of the 11 data sets. SourcererCC finished indexing 111K, 442K, and 1.7M methods within 8 minutes, 32 minutes, and 2 hours respectively. For the complete BigCloneBench (4.8M methods, 365 MLOC), SourcererCC used 6 hours to complete the indexing. Siamese finished indexing the same data sets within 24 minutes, 1 hour 13 minutes, and 5 hours respectively. For the complete BigCloneBench, Siamese took 18 hours 13 minutes to finish the indexing. Since Siamese derives multiple code representations from given code fragments, its indexing time took around three times longer than SourcererCC.

We have also investigated the storage space of Siamese's and SourcererCC's index on the BigCloneBench data set. We found that Siamese's index size is 44 GB while SourcererCC's index size is 3 GB. The large difference in the index size is because of three factors. First, SourcererCC only keeps rare tokens in its index, hence the index size is much smaller than storing all the code tokens as in Siamese. Second, Siamese keeps *n*-gram tokens, instead of raw code tokens, in its index, hence the index size is larger than just storing raw code tokens because there are some duplicated code tokens among the *n*-grams. For example, two consecutive 4-grams `[public|static|int|binarySearch1]` and `[static|int|binarySearch1|(]` share 3-gram of `static|int|binarySearch1`). Third, Siamese stores the *n*-gram tokens for four code representations, hence

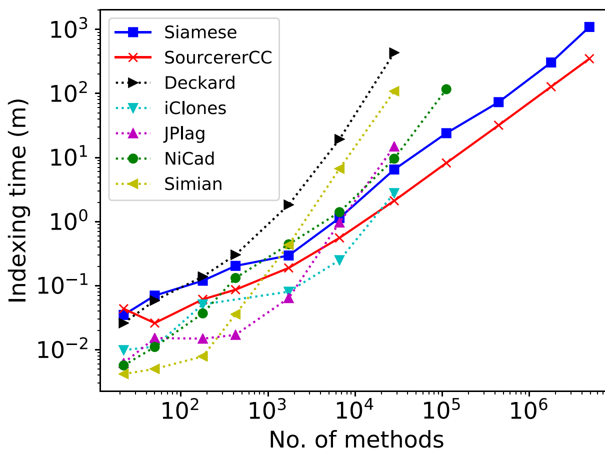


Fig. 8 Indexing time (minutes)

it also occupies four times more space than just one index. Thus, we can see that the Siamese's index size is roughly five times larger than the BigCloneBench data set (which is approximately 9 GB).

For the querying phase, we compare Siamese only to SourcererCC because it is the only tool besides Siamese that successfully scaled to the full BigCloneBench data set. Moreover, it also works in a two-phase approach of indexing and querying like Siamese. Both Siamese and SourcererCC were configured with their default configurations, and only methods with at least ten lines were considered. After each subset was indexed into Siamese's and SourcererCC's index, we performed 100 queries and measured the query response time. We use a fixed set of 100 randomly selected files from BigCloneBench as the queries. One query was issued at a time and the average query time was derived from the execution time of all the queries as shown in Fig. 9. We observed a sharply increasing query response time from SourcererCC when the number of methods in the index grew. Since SourcererCC is designed for detecting clones within a data set, it has its optimal speed when a large collection of files is given as an input and is processed in a batch. Nevertheless, SourcererCC does not respond fast when it comes to a single query because it has to load the index into memory every time a query is issued. On 111K; 442K; and 1.8M methods in the index, SourcererCC's query took 3.4, 9.3, and 28.3 seconds on average. Siamese offered a slightly increasing query response time of 2.5, 3.2, and 5.0 seconds on the same sets of 111K; 442K; and 1.8M methods. On the full BigCloneBench data set with 4.9M methods, Siamese's query time increased slightly to 8 seconds while it took SourcererCC 60 seconds to return the results. This shows that Siamese is suitable for situations where one query is issued at a time, such as searching for code examples, finding similar code candidates for program repairs, or checking for cloned code from the Internet.

To answer RQ4, Siamese offers higher scalability than traditional clone detectors including Deckard, iClones, JPlag, NiCad, and Simian. It scales to a large code corpus of 4.9M methods with 365M SLOC in less than a day. Its indexing time is slower than SourcererCC, but it offers a faster query response time within 8 seconds. Siamese's query response time is marginally affected by the index size. We observed 3 seconds increment in the query time even when the index size grew three times larger.

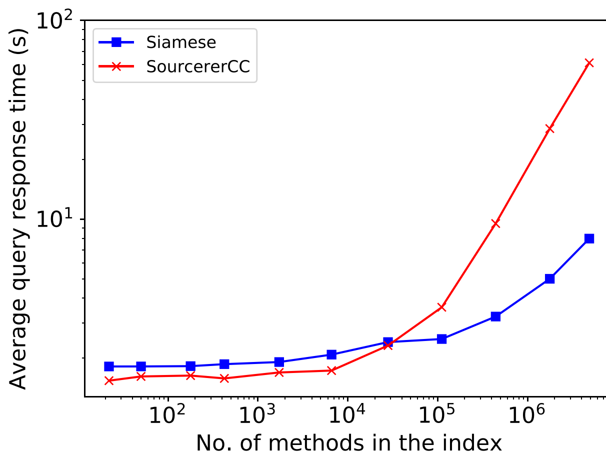


Fig. 9 Querying time (seconds)

6.5 RQ5: Incremental Updates

How fast is Siamese's incremental update?

We followed the same approach used by Hummel et al. (2010) to evaluate the incremental update capability of Siamese. We instructed Siamese's to update versions of software projects in an index of 130,719 GitHub Java projects and measured the time taken to complete the task. To create the code base of GitHub Java projects, we downloaded projects that received at least one star to avoid trivial repositories. We obtained 130,719 projects ranging from 29,465 stars to 1 star in January 2018. The most-starred project is RxJava (29,465 stars), followed by java-design-patterns (27,578 stars) and Elasticsearch (27,385 stars).

We simulated the scenario of maintaining a Siamese GitHub search index when the top three most-starred projects have a new version release. We started by adding all the 130,719 Java projects into Siamese index one project at a time at method-level using incremental addition with the minimum method size of 10 lines (the preferred size of clone detection in large-scale code corpora Sajnani et al. 2016). The indexing took two weeks to finish, and the complete GitHub index contained 8.7 million code fragments with the size of 62 GB.

Then, we downloaded all the available releases of RxJava, java-design-patterns, and Elasticsearch to perform incremental version updates. We choose the three most-starred projects due to their popularity which reflects their chance of being searched for code. As displayed in Table 14, the number of releases and the size of each project varied. Elasticsearch had the highest number of 214 releases, followed by RxJava (153), and java-design-patterns (13) and also had the biggest size on average (62 megabytes), followed by java-design-patterns (15 megabytes) and RxJava (7 megabytes).

For each project, we repeated the process of updating the project's releases from the oldest to the newest version by performing deletion of the current existing release stored in the index followed by addition of the next release to the index. For each update (i.e., deletion/addition) made to the Siamese's index, we measured the time required to finish the task. The results are shown in Fig. 10. The average time of updating java-design-patterns, which was the smallest of the three projects, took 6.6 seconds on average (median 6.5s, max 8.2s). For RxJava, the average project update time was 17 seconds (median 12.8s, max 51.1s). For Elasticsearch, which was the biggest projects and had the largest amount of revisions, the time Siamese took to update the index varied from 20 seconds to approximately 2 minutes with the average of 73 seconds (median 82.1s). The results show that Siamese's incremental update could save the time to prepare the search index of 130,719 GitHub projects when a new version appears from 40,320 minutes (2 weeks) to 2 minutes.

To answer RQ5, Siamese incremental update efficiently handles the changes in software releases and dramatically decreases the index preparation time.

Table 14 GitHub projects used for incremental update

Project	#Releases	Average (Min, Max)		
		Size (MB)	Files	SLOC
RxJava	153	7 (0.4, 16)	582 (1, 1.5K)	82K (3, 244K)
java-design-patterns	13	15 (11, 18)	787 (479, 989)	15K (192, 26K)
Elasticsearch	214	62 (10, 145)	3.7K (1.2K, 5.6K)	399K (87K, 720K)

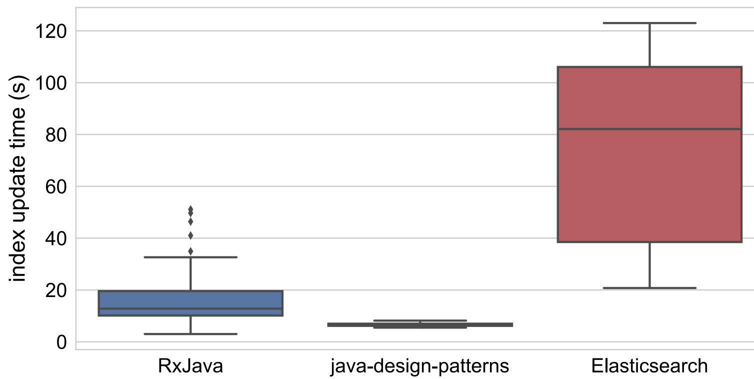


Fig. 10 Incremental index update time

7 Case Studies

Siamese is a general clone search tool which is applicable for several tasks involving code similarity measurement. This section discusses the applications of Siamese to facilitate software development and research with two use cases of online code clone detection and clone search with automated license analysis.

7.1 Online Code Clone Detection on Stack Overflow

Recent studies (Abdalkareem et al. 2017; Acar et al. 2016; An et al. 2017; Baltes and Diehl 2018) show that developers are often reusing code snippets from Stack Overflow in real software projects. We foresee that, in the future, it is important that clone detection will not only happen between local software projects, but also between software projects and online sources, such as Stack Overflow Q&A websites. We call such clones *online code clones*. Compared to traditional code clones between software systems, online code clones are harder to locate and fix since the search space in online code corpora is much larger and no longer confined to a local repository. Hence, the clone detection technique must be scalable and accurate to locate clones in large-scale online code corpora.

In this case study, we empirically show that researchers can use Siamese to tackle the challenges of online code clone detection. We replicated the study of cloned code snippets between Stack Overflow Java accepted answers and the Qualitas corpus (Ragkhitwetsagul et al. 2019) using Siamese, and compared the results to the existing clone results by Simian and SourcererCC.[8] We used the same data sets of 72,365 Stack Overflow Java code snippets and 111 open source Java projects in Qualitas (shown in Table 15), and followed the same experimental framework to detect online code clones. The Stack Overflow code snippets were extracted from accepted answers using `<code>...</code>` tags. Moreover, we filtered out irrelevant code snippets that were not written in Java by using regular expressions and manual checking. The Qualitas projects were used as-is without any pre-processing. We did not need to partition the clone detection into multiple runs as we previously did

[8]We had also tried NiCad, CCFinderX, iClones, DECKARD, and PMD-CPD, but they failed to analyse incomplete code snippets on Stack Overflow or took too long to report clones. Simian and SourcererCC also have the benefit of having two different clone granularity levels. They complement each other as SourcererCC's clone fragments are always confined to method boundaries while Simian's fragments are not.

Table 15 Stack Overflow and Qualitas datasets

Data set	No. of files	SLOC
Stack Overflow	72,365	1,840,581
Qualitas	166,709	19,614,083

with Simian thanks to the scalability of Siamese. The Qualitas corpus was added to Siamese index and the Stack Overflow code snippets were used as the queries. The search configurations are shown in Table 16. We configured Siamese to consider methods with at least ten lines for the search, which resulted in 71,348 queries out of 149,664 methods in the 72,365 Stack Overflow snippets. We limited the result size to at most 100 code snippets per query.

7.1.1 Similarity Threshold

Siamese is a clone search engine which returns a ranked list of clones based on relevance scores between the query and the retrieved code fragments. The original Siamese has no cut-off threshold to decide whether a retrieved code fragment is a cloned fragment of the query or not. This is desirable behaviour for a search engine because the user will look at only the top n results, but not for a clone detector that the user wants a comprehensive list of clones. To be able to compare the clone results of Siamese to Simian's and SourcererCC's, we adapted Siamese to incorporate a similarity measure called n-gram token ratio as the clone similarity threshold.

***N*-gram Token Ratio (NTR)** is an n-gram based similarity measure specifically invented for Siamese. It is applied during search time. Siamese forces an NTR similarity score based on the number of tokens in the query that match with tokens in the indexed code fragments. It is similar to Jaccard similarity on n-gram tokens, except that the similarity score is purely based on the query tokens instead of a union of tokens from the two code fragments. An NTR similarity score between a query Q and a code fragment F is computed as follows.

$$\mathrm{Sim}_{\mathrm{NTR}} = \frac{|\mathrm{T}_Q \cap \mathrm{T}_F|}{|\mathrm{T}_Q|} \tag{6}$$

where T_Q represents a set of n-gram tokens in Q and T_F represents a set of n-gram tokens in F. Since Siamese uses four code representations for its clone search, the similarity score is applied to each of the four representations. Given a similarity threshold, Siamese retrieves only code snippets that offer an NTR score equal to or higher than the threshold on all four representations. The NTR score is applied when the search is performed. Only code fragments that contain enough tokens to reach the defined NTR similarity threshold are retrieved. This method effectively prunes unrelated code fragments and results in fast query response time. In addition, the NTR is a simple token-based and language agnostic similarity measure. Thus, it supports an analysis of any programming language and also works with incomplete code fragments.

Table 16 Siamese execution on Stack Overflow and Qualitas corpus

Snippets	Result size	Exec. time	Per query	Clone pairs (80% sim.)
72,365	100	1h 55m	0.10s	1,088

7.1.2 Results

As shown in Table 16, Siamese with NTR (Siamese-NTR) took approximately 2 hours to complete the clone detection, and the average clone search time per query is 0.10 seconds. We set the similarity threshold at 80% to be similar to the setting of SourcererCC's clone similarity and obtained 1,088 clone pairs. We also performed an additional analysis regarding the chosen result size of at most 100 snippets. We found that, from 72,365 Stack Overflow Java code snippets and 111 Qualitas projects, there were 382 Stack Overflow snippets that had at least one clone candidate in Qualitas. The average number of clones per query is 0.015, which is very low. There were 381 queries that had only 1 to 49 clone results, and there was only 1 query with 100 clone results (the query is a boiler-plate code snippet of `DocumentListener()` initialisation that can match with a large number of code fragments in Qualitas that involve Java Swing components). Thus, the result size of 100 was large enough to exceed the number of clones per query between the two data sets. Then, we compared the clone candidates to the existing clone results reported by Simian and SourcererCC (denoted SM-SCC) in our previous study (Ragkhitwetsagul et al. 2019). To find common clones between the new results from Siamese and the existing 2,289 SM-SCC clone pairs, we employed the clone matching method used in our previous study by applying the Bellon's ok-match clone agreement with the threshold t of 0.5.

Common Clone Pairs The comparison results are displayed in Fig. 11. There were 413 common clone pairs between the SM-SCC results and Siamese-NTR. The common pairs spread across several clone patterns of QS, SQ, EX, UD, BP, IN, and NC as shown in Table 17. Siamese-NTR reported 125 Qualitas→Stack Overflow (QS) clone pairs out of the 153 discovered QS pairs by Simian and SourcererCC and one Stack Overflow→Qualitas (SQ) clone pair. It reported 111 external sources→Stack Overflow (EX) pairs, 64 unknown direction (UD) pairs, and 112 boiler-plate (BP) pairs. It did not report any inheritance/interface (IN) or non-clone pair (NC).

Distinct Clone Pairs Siamese-NTR discovered 675 clone pairs that were not found before and also missed 1,876 clone pairs reported by SourcererCC and Simian (see Fig. 11). To gain insights into the clone pairs that were found only by Siamese, we performed a manual investigation. We manually checked 245 randomly selected clone pairs from Siamese-NTR-only, the number of statistically significant sample with 95% confidence level and ±5% confidence interval. The manual clone validation reported 197 true clone pairs and 48 false clone pairs (80% precision).

By applying the 80% NTR similarity to the four code representations, we forced Siamese to discover clones that were strictly similar. However, we did find some interesting clone pairs due to the NTR similarity computation that SourcererCC and Simian could not find. Since the n-gram token ratio is computed based on the number of tokens in the query, we

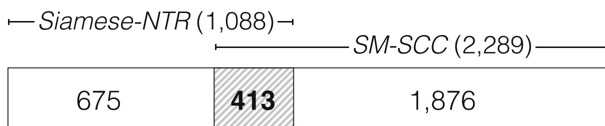


Fig. 11 A comparison of Siamese-NTR clone pairs to the previous results by Simian and SourcererCC (SM-SCC)

Table 17 The 413 SM-SCC online clone pairs that are found by Siamese

QS	SQ	EX	UD	BP	IN	NC
125	1	111	64	112	0	0

found that Siamese-NTR could locate clones of the query inside another method. We call it a *contained* clone pair. An example of a contained clone pair is shown in Fig. 12. The `cmp()` method in the Stack Overflow answer ID 7843663 was reported as clone of the method `sort()` from the `Sorter.java` file from JBoss project although the query matched with only a segment of code inside the `sort()` method. Looking closely into the cloned region between the two clone fragments, we observed a few differences between the two clone fragments. The first clone fragment contained a method called `cmp()` while the second clone fragment contained a method called `compared`, and the first clone fragment contained `if-else` statements while the second clone fragment contained only `if` statements. The Siamese-NTR query could match them because of Type-2 and Type-3 clone representation that allowed variable renaming and added/removed/modified statements. This clone pair was not reported by SourcererCC due to the largely different number of lines. The two code fragments had different sets of rare tokens and the overlap of the rare tokens was possibly smaller than SourcererCC's similarity threshold. Although Simian works at line-level and should not be affected by the size difference between the two code fragments, the clone pair was still not detected, possibly because of the Type-3 differences between `if-else` and `if` statements, which are not supported by Simian.

Moreover, we also randomly looked at 50 clone pairs that were reported by SourcererCC and Simian, but not Siamese, to see why Siamese did not report them. We found many clones that were similar but their similarity probably lower than our defined thresholds of 80%. In a few cases, they were Type-2 clones that were missed by Siamese-NTR. It is because we equally applied 80% similarity to the four representations, and the r_1, i.e., Type-1, representation rejected the clones. In this case, we should give a lower similarity threshold for the r_1 and r_2 representation and only maintain the 80% similarity threshold for r_0 and r_3. Moreover, we observed several Simian-SCC clone pairs that were missed by Siamese because they spanned over multiple methods. They were detected by Simian because Simian only performed line matching to find clones. Since Siamese tried to parse the code into methods when possible, it could not detect this kind of clones.

7.1.3 Discussion

Our replication of online code clone detection between Stack Overflow and Qualitas corpus using Siamese shows that the tool can be applied for fast searching of online code clones. The clone results after applying the *n*-gram token ratio (NTR) similarity measure have some overlaps with the existing clone pairs reported by Simian and SourcererCC and some distinct clone pairs only reported by Siamese and vice versa. We manually checked the pairs reported by Siamese only and found that many of them are true positive pairs. At the same time, Siamese suffers from some false negatives. There were clone pairs that were reported by either Simian or SourcererCC that Siamese could not locate. This is possibly caused by a known problem of tools' configurations (Wang et al. 2013). We leave a detailed investigation as future work.

```
/* query 7843663_0.java */
public int cmp(Object o1, Object o2) {
  int i1 = ((Integer) o1).intValue();
  int i2 = ((Integer) o2).intValue();
  if (i1 < i2) {
    return -1;
  } else if (i1 == i2) {
    return 0;
  } else {
    return 1;
  }
}


/* jboss-5/jboss-5.1.0.GA-src/.../java/Sorter.java, 6, 33 */
public static int[] sort (int[] unsorted) {
  ArrayList list = new ArrayList();
  for (int i = 0; i < unsorted.length; i++) {
    list.add (new Integer (unsorted[i]));
  }
  Collections.sort (list,
    new Comparator() {
    public boolean equals (Object obj) {
      return false;
    }
    public int compare (Object o1, Object o2) {
      int i1 = ((Integer) o1).intValue();
      int i2 = ((Integer) o2).intValue();
      if (i1 < i2) {
        return -1;
      }
      if ( i1 == i2 ) {
        return 0;
      }
      return 1;
    }
  } );
  int[] sorted = new int[unsorted.length];
  for (int i = 0 ; i < list.size() ; i++) {
    sorted[i] = ((Integer) list.get (i)).intValue();
  }
  return sorted;
}
```

Fig. 12 A contained Type-3 clone pair reported by Siamese-NTR

7.2 Clone Search with Software License Analysis

This section illustrates an example of using Siamese for a large-scale exploratory study of clones that are shared between Stack Overflow and GitHub repositories and their license compatibility. This case study is motivated by several discussion threads on StackExchange and Stack Overflow Meta expressing the developers' concerns of license violations by cloning code snippets from Stack Overflow, such as meta.stackexchange.com/questions/12527, meta.stackexchange.com/questions/25956, and meta.stackoverflow.com/questions/321291. Moreover, several recent studies (Abdalkareem et al. 2017; Acar et al. 2016; Baltes and Diehl 2018) reveal that such code cloning between Stack Overflow and open source

projects, especially the ones on GitHub, does exist and sometimes also causes issues like license violations, degraded software quality, and security vulnerabilities.

An et al. (2017) performed a study of clones between Stack Overflow and 399 Android apps and their ramifications of license incompatibility. Their clone detector (NiCad) did not scale to the full data set and had to be executed in 100 smaller runs. Our study leverages the scalability of Siamese to do a similar study on a larger scale of 16,738 GitHub projects in a single run. The data sets used in this study consists of (1) Java code snippets on Stack Overflow and (2) Java source code in GitHub projects. The statistics of the two data sets are shown in Table 18. For GitHub, we used the 16,738 GitHub Java projects with at least ten stars that we used previously in RQ2. For Stack Overflow, we reused the 72,365 extracted code snippets from Java accepted answers employed in the previous case study.

To be able to check for license incompatibilities similar to the study by An et al. (2017), Siamese was extended to support automatic software license identification using pattern matching,[9] so that a manual investigation of software licenses is reduced to only the clone pairs that have incompatible licenses. We built a database of software license patterns by studying the list of 33 software license types on GitHub,[10] reading the text in each license statement, and manually preparing the patterns. During an execution, Siamese identifies the software license in a software project using a two-step approach. First, it reads a dedicated license file `LICENSE` or `LICENSE.txt` at the root level of each GitHub project and matches it with the license patterns in its database to detect a license at project-level. Second, Siamese reads a license statement on top of each Java source code file and performs pattern matching of the license at file-level. When there is a conflict between the file-level and the project-level license, Siamese prefers the finer-grained file-level one. If the tool cannot identify the license, it reports unknown to flag that a manual validation is needed.

Moreover, we configured Siamese to apply the *n*-gram token ratio (NTR) similarity of 100% to every query to make sure that we discovered only exact-match clones (Type-1 clones). It would be interesting to investigate the problem with Type-2 and Type-3 clones as well. However, this will possibly introduce threats to the results. By searching for only 100% similar code fragments, we made sure that we studied only exact copies of the code between Stack Overflow and GitHub projects. In the case of license violations, it is very difficult to confirm the direction of code copying because of a generally lack of evidence. Thus, when performing license analysing, we usually prefer precision over recall (i.e., finding a few correct license violations is better than finding a lot of false positives). By focusing on Type-1 clones, at least we could narrow down the scope of potential license violations to be only clones that are exactly similar, which have much higher chance to be copied than Type-2 or Type-3 clones. The results can be considered as a lower bound of the actual number of clones between the two locations.

Since Siamese supported incremental indexing, we sequentially indexed the 16,738 projects one at a time. This also facilitated the project-based license identification that each project had to be analysed individually. The Siamese index, after analysing all the projects, contained 2,639,565 methods with an index size of 25.6 gigabytes. The indexing with license identification of GitHub projects took one day and twelve hours.

[9]We also tried integrating Ninka (German et al. 2010), a license identification tool, into Siamese but found that it dramatically slowed down the indexing and querying time.

[10]GitHub license type: https://help.github.com/articles/licensing-a-repository

Table 18 The data sets in the case study

Data set	Files	SLOC
Stack Overflow	72,365	1,840,581
GitHub	1,193,478	106,481,517

In the query phase, each code snippet from Stack Overflow was used as a query with a results size of 100. The search for clones with similarity computation between the two data sets took 1 hours and 57 minutes to complete.

7.2.1 Results

We initially set the minimum of 10 lines for clone size since it was recommended for a large-scale clone detection to get rid of trivial clones (Sajnani et al. 2016). With the minimum of 10 lines, we retrieved a large number of clone candidate pairs. However, after manually investigating a few sampled clone pairs, we still found several trivial clones such as `equals` methods or generated GUI-related code. These trivial clones had the size of around 10 to 20 lines, so we increased the minimum clone size to 20 lines. With the larger minimum clone size, 378 clone pairs were reported. This is the lower bound of the number of clone candidate pairs since we might also get rid of true positive clone pairs that were smaller than 20 lines. Nonetheless, as previously discussed, false negatives (i.e., not reporting a clone pair while it is actually a clone pair) are preferred over false positives (i.e., reporting a clone pair while it is actually a non-clone pair) in this case of license violation checking.

We compiled a list of 10 projects having the highest number of clones within the 16,738 projects we analysed as shown in Table 19. The Google's J2ObjC (4,981 stars), which is a command-line tool that translates Java to Objective-C code, has the highest number of 17 clone pairs. The second is JavaExercises project (34 stars), which contains a lot of Java programming examples, with 13 clone pairs followed by XobotOS, Android porting from Java/Dalvik to C#, (1,278 stars) with 11 clone pairs; JalaliCalendar, a Java Persian calendar library, (51 stars) with 9 clone pairs; guideshow (85 stars – 7 pairs); react-native-lanscan (16 stars – 7 pairs); AOSP Framework Support Library (1,253 stars – 5 pairs); java-tool (16 stars – 5 pairs); Dropbox's hackpad (3,085 stars – 5 pairs); and AndroidRAT (37 stars – 5 pairs). We did not confirm the direction of cloning. However, after looking at the numbers,

Table 19 GitHub projects with the highest no. of clones

Project name	Stars	Clone pairs
google/j2objc	4,981	17
biblelamp/JavaExercises	34	13
xamarin/XobotOS	1,278	11
amirmehdizadeh/JalaliCalendar	51	9
javajavadog/guideshow	85	7
Odinvt/react-native-lanscan	16	7
aosp-mirror/platform_frameworks_support	1,253	5
osglworks/java-tool	16	5
dropbox/hackpad	3,085	5
ibrahimbalic/AndroidRAT	37	5

we observed an interesting patterns: high and low stars projects both have high numbers of clones, which possibly indicate the direction of cloning. We left an investigation for future work.

Siamese reported the same license for 131 pairs, and different license for 247 pairs. We further analysed the licenses in the clone pairs and the results are displayed in Table 20. For the same license, 127 clone pairs do not have a license statement and 4 pairs have the Apache-2.0 license. On the other hand, 65% of the clone pairs with different licenses (247 out of 378) contain no license on Stack Overflow while having a license on GitHub. The three highest number of clone pairs have: 1) no license on Stack Overflow but Apache-2.0 license on GitHub (139 pairs); 2) no license on Stack Overflow and GPL-2.0 license on GitHub (32 pairs); and 3) no license on Stack Overflow but MIT license on GitHub (27 pairs).

Although we did not confirm the violations of software license, the findings from the study show that we can use Siamese to locate potential candidates of clones with software license incompatibility, which save the time for a human investigator.

7.2.2 Discussion

The study demonstrated an application of Siamese to efficiently and effectively find clones which potentially violate software licenses. Siamese found a number of clone pairs between Stack Overflow and GitHub projects that the code were exactly matched but had different software licenses. These clone pairs with different licenses may or may not create

Table 20 License comparison of the clones

License	Stack Overflow	GitHub	Frequencies
Same license	None	None	127
	Apache-2.0	Apache-2.0	4
Total			*131*
Different license	None	Apache-2.0	139
	None	GPL-2.0	32
	None	MIT	27
	None	GPL-3.0	12
	None	Apache	10
	None	BSD-2-Clause	6
	None	BSD-3-Clause	5
	None	LGPL-3.0	5
	None	AGPL-3.0	4
	None	Artistic-2.0	2
	None	Unknown	2
	None	CC0-1.0	1
	Unknown	GPL-2.0	1
	Unknown	WTFPL	1
Total			*247*
Grand total			*378*

licensing conflicts depending on the direction of cloning, which requires a further thorough investigation and is beyond the scope of this paper.

8 Threats to Validity

8.1 Internal and Construct Validity

We compared the tools' performance based on three standard measurements of precision at 10, MAP, and MRR from information retrieval. Nevertheless, in some situations, other measurements may be required and might not produce the same results. We compared Siamese to seven state-of-the-art clone detectors on the default and the optimal configurations. We might not cover all the tools' parameters due to our selection of the value ranges and the increasing step sizes. Moreover, we may introduce bias when comparing code clone detection tools that return the lines or tokens in the clone (e.g., CCFinderX, Simian), to Siamese which only returns method-level or file-level clones. The query reduction thresholds were derived from the Bellon corpus and may be subjective to the clones in the corpus but we mitigated the issue by avoiding using Bellon corpus in the evaluation data sets to avoid configuration bias. We confirmed the findings with the Qualitas corpus and observed the same result. The manual validation of clone search results was carefully performed but may still be subject to manual judgement and human errors. The MRR and precision at 10 used to measure Siamese's precision are based on the top n clone results and may not reflect the precision score, which is based on the total number of returned results. The comparison of Siamese to three code search tools, FaCoy, Searchcode, and Krugle, is based on a different code search index, thus there may be bias in the reported search results. The proposed clone search technique do not consider the order of n-grams and this decision allows Siamese to detect relocated code statements. It is possible that Siamese may report some false positives with the presence of two non-cloned code fragments sharing 100% n-grams. However, based on our empirical observations, n-grams already partially capture the sequences in the code. Thus, having two code fragments that are not clones but completely share 100% n-grams are very unlikely. The clones found in the case study are subject to the chosen similarity measure and the threshold. The results may be different if other code similarity technique is selected. We used code clone detection benchmarks to evaluate Siamese on code clone search. This could possibly introduce some bias due to such benchmarks being highly populated with clones compared to real-world scenarios. Lastly, the clones with incompatible licenses are based on our manually-prepared software license patterns, which may produce false positives or false negatives.

8.2 External Validity

The three error measures, precision at 10, MAP, and MRR are based on queries with known relevant results and may not fully represent real-world queries. We carefully chose the data sets for our experiments and Siamese was evaluated on multiple data sets to cover several types of cloned code and to alleviate the evaluation bias. Nonetheless, some of them are generated data sets and may not fully cover the characteristics of code clones in real world software. We tried to mitigate the issue by evaluating Siamese on 16,738 real-world Java projects from GitHub, similar to what the authors of the FaCoy study have done. Our multi-representation, query reduction techniques, indexing and searching performance of Siamese are evaluated with Java and may not be generalised to other languages. However,

we designed the Siamese's architecture to work with other programming languages by plugging in a new tokeniser and code normaliser module. The indexing and querying performance of Siamese and SourcererCC were measured on a single computer and may not represent their performance on other computers with different specifications or a cluster of multiple Elasticsearch instances. The criteria for selecting the GitHub projects for incremental update is based on the stars and may not be generalised to other Java projects. The results from the first case study are based on 72,365 Java code snippets on Stack Overflow and 111 Qualitas projects and may not be generalised to other programming languages or other software projects. Similarly, the results from the second case study are restricted to 16,738 GitHub Java projects with at least ten stars. The results may be different for projects with lower popularity and also projects in other programming languages. Moreover, we did not consider Type-2 and Type-3 clones to minimise the number of false positives. The reported number of clones and their license analysis may not cover code clones with modifications.

9 Related Work

9.1 Code Clones

Code clones, which are similar code fragments, occur by programmers duplicating source code with or without modifications (Roy et al. 2009). It is a common activity found in software development, and amount of clones may be used as a proxy to measure the software quality (Fowler 1999). Clones may or may not complicate software maintenance depending on several contexts, such as the languages and types of software projects being analysed (one version vs. multiple variations of hardware drivers) (Kapser and Godfrey 2006) or how consistently are the changes applied to clones (Aversano et al. 2007; Juergens et al. 2011). Nonetheless, clone researchers agree that clones need to be made explicit so that an appropriate clone management process can be carried out (Chatterji et al. 2016).

The intention behind code cloning can vary from unintentional use of coding idioms (Kapser and Godfrey 2006) to reusing well-written code in order to preserve functionality and performance (Kamiya et al. 2002). Software development industry has utilised code cloning intensively. Roy et al. (2009) and Davey et al. (1995) reported that a substantial percentage (7–23% and 20–30% respectively) of a software module contains clones. Baxter et al. (1998) similarly found that in average 12.7% are clones in the commercial software project used in their study.

There are various approaches to code clone detection including text-based (Harris 2015; Roy and Cordy 2008), token-based (Kamiya et al. 2002; Prechelt et al. 2002; Sajnani et al. 2016; Schleimer et al. 2003), tree-based (Baxter et al. 1998; Jiang et al. 2007), graph-based (Krinke 2001), or deep learning techniques (Li et al. 2017; White et al. 2016) to locate clones within or between software projects. Nevertheless, with the growing amount of source code on the Internet, programmers are no longer limited to cloning code from local software repositories, but are allowed to reuse a vast amount of code online (Acar et al. 2016; Yang et al. 2017; Abdalkareem et al. 2017). A recent large-scale study by Lopes et al. (2017) shows that 70% of the code on GitHub are clones.

9.2 Scalable Clone Detection

Hummel et al. (2010) is among the first to present clone detection tool for Type-1 and Type-2 clones that is incremental and scalable using index-based techniques. A clone index is

created from source code sequence hashes. The tool evaluation shows that it returns clones for a file in 42M SLOC Eclipse code base within 1 second. The tool can be distributed to gain even higher scalability. Lavoie et al. (2010) propose a new version of a dynamic programming algorithm called *DP-matching* and use it for clone fragment similarity calculation on a graphic processing unit (GPU). However, the evaluation results show that their GPU-based approach only slightly increases the performance of DP-matching from its CPU implementation. Livieri et al. (2010) present a scalable approach for clone detection using *n*-gram matching. Their evaluation of a tool implementing the idea, called *Yocca*, shows that it is more scalable than CCFinder and Simian. However, the authors only discuss scalability and did not report the clone detection accuracy of the tool.

Inoue et al. (2012) propose a system called *Ichi Tracker* that leverages the power of three code search engines: Google Code Search, Koders,[11] and SPARS/R.[12] The system is designed for tracking an origin and evolution of source code. Nevertheless, the Google Code Search and Koders are no longer available, which severely affects the usability of the system. Koschke (2014) presented a scalable inter-system clone detection using a suffix-tree-based algorithm. The author evaluated the use of index-based hashes of token *n*-grams to speed up the clone detection process and concluded that building an index was worthwhile only if it is reused multiple times. Moreover, he showed that software metrics and a learned decision tree increase the clone detection's precision. Ohmann and Rahal (2014) propose a large-scale approach, called *Program It Yourself (PIY)*, for efficient source code plagiarism detection using parallel execution and clustering algorithms. PIY uses *n*-grams to create vectors and compare them using Manhattan and cosine distance metrics. Its efficiency in large-scale data is dramatically enhanced by including parallel execution and clustering methods. However, the biggest dataset tested with PIY contains approximately 23,000 files which is still relatively small compared to BigCloneBench.

Tamersoy et al. (2014) show an efficient approach for large-scale malware detection based on association graphs. The authors propose a method to estimate machine-program co-occurrence strength using MinHashing algorithm (Rajaraman and Ullman 2011) and Locality Sensitivity Hashing (LSH) (Slaney and Casey 2008), and implement a tool called *AESOP*. The study analyses the massive amount of data from the Symantec Norton's Community Watch containing 11 million machines and 43 million files. Nonetheless, based on the underlying approach of AESOP, the tool needs source code data that contain associations between the code and their owners, which may not always exist. Keivanloo et al. (2014) presents a code search system aiming to find working code examples. It tackles the problem of current code search systems that rely on API names as search keywords by proposing the *abstract programming solution* extraction approach. Its evaluation on IJaDataset 2.0 shows that the approach outperforms an industrial Ohloh Code search engine on finding working code examples. However, the query set in the evaluation is limited to only 15 queries, and the comparison of the two systems has been performed on a different data set, which makes the findings not generalised. Svajlenko et al. (2014) present a large-scale clone detection solution by utilising classic clone detectors. The authors introduce a scalable non-deterministic algorithm called *shuffling framework*. The framework partitions the dataset into small subsets that fit with the tool's input size and environments. The experimental results show that the framework can enable Simian and NiCad to execute against large datasets. However, the

[11] http://code.openhub.net

[12] http://sel.ist.osaka-u.ac.jp/SPARS/index.html.en

proposed system suffers from problems of line-counting mismatches between the framework and each clone detection tool, high generation time of inverted index, and a bottleneck from sequential subset generation.

Sajnani et al. (2016) create a scalable code detection tool called *SourcererCC*. The tool is a token-based detector based on an optimised inverted index to scalably retrieve clone pair candidates within a short amount of time. The authors incorporate two filtering heuristics, sub-block overlap and token positions, to dramatically reduce the number of pairwise comparisons. The tool report high recall and precision compared to other state-of-the-art clone detectors. It also scales to the IJaDataset 2.0 with 250 million lines of code. Later, Nishi and Damevski (2018) extended SourcererCC by adding adaptive prefix filtering to obtain higher clone detection scalability. Svajlenko and Roy (2017) adopted Sajnani et al. (2016)'s sub-block heuristic into their scalable clone detector, *CloneWorks*. The tool's scalability is enhanced using partitioning of input code fragments to fit within an allowed memory limit. They use a slightly modified version of Jaccard similarity to detect clones. CloneWorks offers high precision and recall of Type-1, Type-2, and Type-3 clones on BigCloneBench compared to iClones, NiCad, and SourcererCC while giving a much faster detection speed than SourcererCC. The tool finishes its clone detection in 4 hours (conservative configurations) and 10 hours (aggressive configurations) compared to 110 hours by SourcererCC.

Oreo is a scalable clone detector created by Saini et al. (2018) that integrates deep learning, information retrieval, and software metrics. By training a deep neural network model on 24 software metrics of cloned and non-cloned pairs reported by SourcererCC from 50,000 GitHub projects, the tool is capable of detecting a large number of challenging Type-3 and Type-4 clone pairs. Oreo completes a clone detection on IJaDataset 2.0 within about a day.

9.3 Code Search

Internet-scale code search is an emerging field of research that targets on finding source code data on the Internet for code reuse, bug fixing, or program comprehension (Gallardo-Valencia and Sim 2009).

There are several tools available for code search. One can use Google as a code search engine by choosing desired functionalities as the keywords (Sim et al. 2011). There are also dedicated code search engines such as Krugle, searchcode, Codata, or Black Duck Open Hub Code Search (formerly known as Koders) that take programming language structure into account while searching. Researchers also create code search techniques and tools for their study, which some of them are later opened for free of use.

Linstead et al. (2009) invented *Sourcerer*, a source code retrieval system on the Internet-scale with million lines of code. Bajracharya et al. (2010) use structural semantic indexing (SSI) to return code examples based on similarity of API usage. The evaluation of 346 jars from Eclipse framework shows that SSI-based schemes are preferred over baseline schemes that do not include usage similarity in the search. Martie et al. (2017) reflect that code search is an iterative process where information seekers need to keep adapting their search queries until they find relevant results. They present two tools,*CodeLikeThis (CLT)* and *CodeExchange (CE)*, to facilitate iterative code search and perform a user study to show that the tools could improve code search experience. Niu et al. (2017) improve the ranking schema of code results by applying a learning-to-rank machine learning algorithm. They find that the approach outperforms five existing ranking schemas on normalised discounted cumulative gain (NDCG) by at least 35.65%. The work by Gu et al. (2018) uses deep learning techniques called *CODEnn* (Code-Description Embedding Neural Network) to match

code snippets and natural language descriptions in the query using joint high-dimensional embedding vectors.

We refer the readers to a book by Sim and Gallardo-Valencia (2013) which presents a comprehensive list of code search studies including motivation and behaviours of programmers to search for code, a user study on Internet-scale code search, and the infrastructures and techniques for code and software component search engines.

9.4 Code Clone Search

In this paper, we focus on a specific kind of code search called *code clone search*. Code clone search is a special case of code search where a piece of code is given as a query instead of natural text keywords. By executing the query, a clone search system returns a list of clones of the query. Code clone search differs from code clone detection because it is query-centric. Instead of looking for a complete set of clone pairs or clone groups in given code corpora as in clone detection, a clone search tool retrieves only clones that are associated with the query. Due to the similarity between code clone detection and clone search and the limited number of clone search tool available, sometimes clone detectors are also used to search for similar code. Here, we discuss techniques that are dedicatedly invented for clone search.

Lee et al. (2010) search for clone using structural similarity based on R*Tree indexing structure. The technique searches for clones within 492 open source projects with less than a second. *Exemplar* (Grechanik et al. 2010) leverages program analysis with information retrieval to search for highly relevant applications. The tool searches for similar applications based on similarity of their API calls. A user study with 39 professional Java programmers showed that Exemplar outperformed SourceForge in searching for relevant applications. *Portfolio* (McMillan et al. 2011) uses multiple techniques including natural language processing, PageRank, and spreading activation network to find relevant functions and projects. Keivanloo et al. presented a real-time code clone search which utilises ontologies to expand the search keywords (Keivanloo et al. 2012). The authors also present other variations of real-time clone search system using multi-level indexing (Keivanloo et al. 2011a, b), and abstract programming solutions (Keivanloo et al. 2014). Ishio et al. (2017) present a scalable approach for detecting clone-and-own software packages using b-bit minwise hashing technique. Then, an aggregated file similarity is applied to rank the returned search components. The technique gives a recall score of 0.907 in an evaluation. Kim et al. (2018) created *FaCoy*, a code-to-code search system that leverages the information on Stack Overflow to expand the keywords in the search query. The tool aims for searching semantically similar code. The evaluation shows that the technique can return code snippets with similar runtime behaviours to the query snippet and are useful for patch recommendation.

9.5 Query Quality Improvement

Information retrieval community has proposed techniques to increase the quality of queries before submitting them to a search engine by reducing the size of long queries (Kumaran and Allan 2007; Kumaran and Carvalho 2009; Balasubramanian et al. 2010; Bendersky and Croft 2008; Robertson 1990). Kumaran and Allan (2007) propose a method to select short sub queries based on mutual information, maximum spanning tree, and named entities. Bendersky and Croft (2008) derive key concepts from long queries to improve retrieval effectiveness. Kumaran and Carvalho (2009) and Balasubramanian et al. (2010) adopt learning to rank, a machine learning technique, with nine query quality predictors in order to

choose a sub query with an optimal size. Zhang et al. (2017) improve natural-language queries for code search by looking for additional search keywords from semantically related API class names. Our study is the first to present a code query reduction technique based on token document frequencies.

10 Conclusion

We present the architecture of a scalable and incremental clone search and its implementation as a tool called Siamese. Siamese incorporates a novel multiple code representations (MR) technique to transform Java code into four code representations to detect different types of clone at once and a query reduction (QR) technique to automatically reduce the query size on-the-fly based on token document frequencies. We showed that MR-QR increased clone search precision compared to the baseline of text search engine. We evaluated Siamese on three data sets: OCD (clones with obfuscation, compilation, decompilation), SOCO (clones with boiler-plate code), and BigCloneBench (a clone benchmark with 8 million clone pairs). Siamese offers 95% and 99% mean average precision on the OCD and the SOCO data set respectively and also offers high recall for all clone types in the BigCloneBench data set. When compared to other three code search engines on the 10 highest-voted Stack Overflow code snippets, Siamese returns the largest number of Type-3 clones. Furthermore, Siamese provides scalability by returning clone search results in less than 8 seconds even on the largest data set of 365 million lines of code. The technique supports incremental index updating that allows fast update to the existing index without the need to recreate the index from scratch. The two case studies illustrate the applications of Siamese to software engineering research. They show that Siamese can be adapted to the problem of online code clone detection and software license analysis. The Siamese clone search approach opens possibilities for discovering online code reuse, finding similar code examples, detecting software plagiarism, and finding software licensing conflicts in real time on large-scale code corpora.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

Abdalkareem R, Shihab E, Rilling J (2017) On code reuse from StackOverflow: an exploratory study on android apps. Inf Softw Technol 88:148–158

Acar Y, Backes M, Fahl S, Kim D, Mazurek ML, Stransky C (2016) You get where you're looking for: the impact of information sources on code security. In: SP '16, pp 289–305

An L, Mlouki O, Khomh F, Antoniol G (2017) Stack Overflow: a code laundering platform? In: SANER '17, pp 283–293

Aragon Consulting Group Inc (2018) Krugle. http://krugle.com, Online; Access 23-April-2018

Aversano L, Cerulo L, Di Penta M (2007) How clones are maintained: an empirical study. In: Proceedings of the 11th European conference on software maintenance and reengineering (CSMR '07), IEEE, Los Alamitos, California, USA, pp 81-90

Bajracharya SK, Ossher J, Lopes CV (2010) Leveraging usage similarity for effective retrieval of examples in code repositories. In: Proceedings of the 18th ACM SIGSOFT international symposium on foundations of software engineering (FSE '10), p 157

Balasubramanian N, Kumaran G, Carvalho VR (2010) Exploring reductions for long web queries. In: SIGIR '10, p 571

Baltes S, Diehl S (2018) Usage and attribution of Stack Overflow code snippets in GitHub projects. Empir Softw Eng:1–37

Bauer V, Volke T, Eder S (2016) Combining clone detection and latent semantic indexing to detect re-implementations. In: Proceedings of the IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER '16), pp 23–29

Baxter I, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: ICSM '98, vol 98, pp 368-377

Beckman NE, Kim D, Aldrich J (2011) An empirical study of object protocols in the wild. In: ECOOP '11, pp 2–26

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. TSE 33(9):577–591

Bendersky M, Croft WB (2008) Discovering key concepts in verbose queries. In: SIGIR'08, p 491

BlackDuck (2016) OpenHub. http://code.openhub.net, online; access 18-May-2016

Boyter B (2018) Searchcode. https://searchcode.com, online; access 23-April-2018

Burrows S, Tahaghoghi SMM, Zobel J (2007) Efficient plagiarism detection for large code repositories. Software: Practice and Experience 37(2):151–175

Chatterji D, Carver JC, Kraft NA (2016) Code clones and developer behavior: results of two surveys of the clone research community. Empir Softw Eng 21(4):1476–1508

Craswell N (2009) Encyclopedia of Database Systems. Springer, Berlin

Davey N, Barson P, Field S, Frank R, Tansley D (1995) The development of a software clone detector. Int J Appl Softw Technol 1:3–4

Elasticsearch BV (2012) Lucene's practical scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html, online; access 20-March-2017

Elasticsearch BV (2016) Elasticsearch. https://www.elastic.co/products/elasticsearch, online; access 25-Jun-2016

Flores E, Rosso P, Moreno L, Villatoro-Tello E (2014) Detection of source code re-use. http://users.dsic.upv.es/grupos/nle/soco/, accessed: 2016-02-14

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley, Boston

Gallardo-Valencia RE, Sim SE (2009) Internet-scale code search. SUITE '09, pp 49–52

German DM, Manabe Y, Inoue K (2010) A sentence-matching method for automatic license identification of source code files. In: ASE '10, p 437

Göde N, Koschke R (2009) Incremental clone detection. In: CSMR'09, pp 219–228

Grechanik M, Fu C, Xie Q, McMillan C, Poshyvanyk D, Cumby C (2010) A search engine for finding highly relevant applications. In: ICSE '10, pp 475–484

Gu X, Zhang H, Kim S (2018) Deep code search. In: Proceedings of the 40th international conference on software engineering (ICSE '18), pp 933–944

Harris S (2015) Simian – similarity analyser, version 2.4. http://www.harukizaemon.com/simian/, accessed: 2016-02-14

Hummel B, Juergens E, Heinemann L, Conradt M (2010) Index-based code clone detection: incremental, distributed, scalable. In: ICSM'10, pp 1–9

Inoue K, Sasaki Y, Xia P, Manabe Y (2012) Where does this code come from and where does it go? — integrated code history tracker for open source systems. In: ICSE '12, pp 331–341

Ishio T, Sakaguchi Y, Ito K, Inoue K (2017) Source file set search for clone-and-own reuse analysis. In: Proceedings of the IEEE/ACM 14th international conference on mining software repositories (MSR '17), pp 257–268

Jiang L, Misherghi G, Su Z, Glondu S (2007) DECKARD: scalable And accurate tree-based detection of code clones. In: ICSE'07. IEEE, Minneapolis, pp 96-105

Juergens E, Deissenboeck F, Hummel B (2011) Code similarities beyond copy & paste. In: Proceedings of the 15th European conference on software maintenance and reengineering (CSMR '11), IEEE, pp 78–87

Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multilinguistic token-based code clone detection system for large scale source code. TSE 28(7):654–670

Kapser C, Godfrey MW (2006) Cloning considered harmful considered harmful. In: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06), Benevento, taly, pp 19-28

Kawaguchi S, Yamashina T, Uwano H, Fushida K, Kamei Y, Nagura M, Iida H (2009) SHINOBI: a tool for automatic code clone detection in the IDE. In: WCRE '09, pp 313–314

Ke Y, Stolee KT, Goues CL, Brun Y (2015) Repairing programs with semantic code search. In: ASE'15, pp 295–306

Keivanloo I, Rilling J, Charland P (2011a) Internet-scale real-time code clone search via multi-level indexing. In: WCRE '11, pp 23–27

Keivanloo I, Rilling J, Charland P (2011b) SeClone – a hybrid approach to internet-scale real-time code clone search. In: ICPC '11, pp 223–224
Keivanloo I, Forbes C, Rilling J (2012) Similarity search plug-in: Clone detection meets internet-scale code search. In: SUITE '12, pp 21–22
Keivanloo I, Rilling J, Zou Y (2014) Spotting working code examples. In: ICSE '14, pp 664–675
Kim K, Kim D, Bissyandé TF, Choi E, Li L, Klein J, Traon YL (2018) FaCoY – a code-to-code search engine. In: ICSE'18
Knuth DE (1971) An empirical study of fortran programs. Software: Practice and Experience 1(2):105–133
Koschke R (2014) Large-scale inter-system clone detection using suffix trees and hashing. Journal of Software: Evolution and Process 26(8):747–769
Koschke R, Falke R, Frenzel P (2006) Clone detection using abstract syntax suffix trees. In: WCRE '06, pp 253–262
Krinke J (2001) Identifying similar code with program dependence graphs. In: WCRE '01
Kumaran G, Allan J (2007) A case for shorter queries, and helping users create them. In: NAACL-HLT '07, pp 220–227
Kumaran G, Carvalho VR (2009) Reducing long queries using query quality predictors. In: SIGIR'09, p 564
Lavoie T, Eilers-Smith M, Merlo E (2010) Challenging cloning related problems with gpu-based algorithms. In: Proceedings of the 4th international workshop on software clones (IWSC '10), ACM, Cape Town, South Africa, pp 25-32
Lee MW, Roh JW, Hwang SW, Kim S (2010) Instant code clone search. In: FSE '10, p 167
Li L, Feng H, Zhuang W, Meng N, Ryder B (2017) CCLearner: a deep learning-based clone detection approach. In: ICSME'17, pp 249–260
Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2009) Sourcerer: mining and searching internet-scale software repositories, vol 18, pp 300–336
Livieri S, German DM, Inoue K (2010) A needle in the stack: Efficient clone detection for huge collections of source code. Tech. rep., OSaka University
Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) DéjàVu: a map of code duplicates on GitHub. Proceedings of the ACM on Programming Languages (OOPSLA) 1:1–28
Manning CD, Raghavan P, Schutze H (2009) An introduction to information retrieval, vol 21. Cambridge University Press, Cambridge
Martie L, Hoek AVD, Kwak T (2017) Understanding the impact of support for iteration on code search. In: ESEC/FSE '17, pp 774–785
McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: finding relevant functions and their usages. In: ICSE '11, p 111
Miller GA (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychol Rev 63(2):81–97
Myles G, Collberg C (2005) K-gram based software birthmarks. In: SAC '05, p 314
Nasehi SM, Sillito J, Maurer F, Burns C (2012) What makes a good code example?: a study of programming Q&A in StackOverflow. In: ICSM'12, pp 25–34
Nguyen TT, Nguyen HA, Al-Kofahi JM, Pham NH, Nguyen TN (2009) Scalable and incremental clone detection for evolving software. In: ICSM '09, pp 491–494
Nishi MA, Damevski K (2018) Scalable code clone detection and search based on adaptive prefix filtering. J Syst Softw 137:130–142
Niu H, Keivanloo I, Zou Y (2017) Learning to rank code examples for code search engines. Empir Softw Eng 22(1):259–291
Ohmann T, Rahal I (2014) Efficient clustering-based source code plagiarism detection using PIY. Knowl Inf Syst 43(2):445–472
Omar C, Yoon YS, LaToza TD, Myers BA (2012) Active code completion. In: ICSE '12, pp 859–869
Park JW, Lee MW, Roh JW, Hwang SW, Kim S (2014) Surfacing code in the dark: an instant clone search approach. Knowl Inf Syst 41(3):727–759
Parr T, Harwell S, Kochurkin I (2017) Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4, accessed: 2017-11-21
Ponzanelli L, Bacchelli A, Lanza M (2013) Seahawk: Stack Overflow in the IDE. In: ICSE '13, pp 1295–1298
Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M (2014) Mining StackOverflow to turn the IDE into a self-confident programming prompter. In: MSR '14, pp 102–111
Prechelt L, Malpohl G, Philippsen M (2002) Finding plagiarisms among a set of programs with JPlag. J Univ Comput Sci 8(11):1016–1038

Ragkhitwetsagul C, Krinke J, Clark D (2018) A comparison of code similarity analysers. Empir Softw Eng 23(4):2464–2519

Ragkhitwetsagul C, Krinke J, Paixao M, Bianco G, Oliveto R (2019) Toxic code snippets on Stack Overflow. Transactions on Software Engineering (Early Access)

Rajaraman A, Ullman JD (2011) Mining of massive datasets, vol 67. Cambridge University Press, Cambridge

Rilling J, Keivanloo I, Forbes C, Erfani M (2018) IJaDataset 2.0. https://sites.google.com/site/asegsecold/projects/seclone, online; access 13-March-2018

Robertson S (1990) On term selection for query expansion. J Doc 46(4):359–364

Roy CK, Cordy JR (2008) NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: ICPC '08, pp 172–181

Roy CK, Cordy JR (2009) Near-miss function clones in open source software: an empirical study. J Softw Maint Evol Res Pract 26(12):165–189

Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci Comput Program 74(7):470–495

Sadowski C, Stolee KT, Elbaum S (2015) How developers search for code: a case study. In: ESEC/FSE '15, pp 191–201

Saini V, Farmahinifarahani F, Lu Y, Baldi P, Lopes C (2018) Oreo: detection of clones in the twilight zone. In: The 26th ACM joint European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE '18)

Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) SourcererCC: scaling code clone detection to big-code. In: ICSE'16, pp 1157–1168

Salton G, Wong A, Yang CS (1975) A vector space model for automatic indexing. Commun ACM 18(11):613–620

Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: local algorithms for document fingerprinting. In: SIGMOD '03, ACM, p 76

Sim SE, Gallardo-Valencia RE (2013) Finding source code on the web for remix and reuse. Springer, Berlin

Sim SE, Umarji M, Ratanotayanon S, Lopes CV (2011) How well do search engines support code retrieval on the web? ACM Trans Softw Eng Methodol 21(1):1–25

Slaney M, Casey M (2008) Locality-sensitive hashing for finding nearest neighbors. IEEE Signal Proc Mag 25(2):128–131

Smucker MD, Allan J, Carterette B (2007) A comparison of statistical significance tests for information retrieval evaluation. In: CIKM '07, p 623

Svajlenko J, Roy CK (2014) Evaluating modern clone detection tools. In: Proceedings of the 30th international conference on software maintenance and evolution (ICSME '14), IEEE, pp 321–330

Svajlenko J, Roy CK (2015) Evaluating clone detection tools with BigCloneBench. In: ICSME'15, pp 131–140

Svajlenko J, Roy CK (2016) BigCloneEval: a clone detection tool evaluation framework with BigCloneBench. In: Proceedings of the international conference on software maintenance and evolution (ICSME '16), vol 1, pp 596–600

Svajlenko J, Roy CK (2017) Fast and flexible large-scale clone detection with CloneWorks. In: Proceedings of the IEEE/ACM 39th international conference on software engineering companion (ICSE-C '17), pp 27–30

Svajlenko J, Islam JF, Keivanloo I, Roy CK, Mia MM (2014) Towards a big data curated benchmark of inter-project code clones. In: ICSME'14, pp 476–480

Tamersoy A, Roundy K, Chau DH (2014) Guilt by association: Large scale malware detection by mining. In: Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining (KDD '14). ACM, New York, pp 1524-1533

Taube-Schock C, Walker RJ, Witten IH (2011) Can we avoid high coupling? In: ECOOP '11, pp 204–228

Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) Qualitas corpus: a curated collection of Java code for empirical studies. In: APSEC '10, pp 336–345

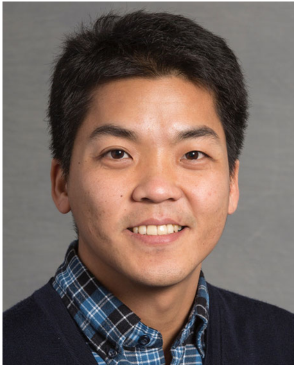
van Bruggen D (2017) JavaParser – process Java code programmatically. http://javaparser.org, accessed: 2017-11-21

Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2 (Summer, 2000)):101–132

Vasilescu B, Serebrenik A, van den Brand M (2011) You can't control the unfamiliar: a study on the relations between aggregation techniques for software metrics. In: ICSM '11, pp 313–322

Wang T, Harman M, Jia Y, Krinke J (2013) Searching for better configurations: a rigorous approach to clone evaluation. In: ESEC/FSE '13, pp 455–465

White M, Tufano M, Vendome C, Poshyvanyk D (2016) Deep learning code fragments for code clone detection. In: ASE '16, pp 87–98
Yang D, Martins P, Saini V, Lopes C (2017) Stack Overflow in Github: any snippets there? In: MSR '17
Zhang F, Niu H, Keivanloo I, Zou Y (2017) Expanding queries for code search using semantically related API class-names. TSE https://doi.org/10.1109/TSE.2017.2750682
Zhang H (2008) Exploring regularity in source code: software science and Zipf's law. In: WCRE'08, pp 101–110
Zipf GK (1932) Selective studies and the principle of relative frequency in language. Harvard University Press, Cambridge



Chaiyong Ragkhitwetsagul is a lecturer at the Faculty of Information and Communication Technology (ICT), Mahidol University, Thailand. He received the PhD degree in Computer Science at University College London, where he was part of the Centre for Research on Evolution, Search, and Testing (CREST). His research interests include code search, code clone detection, software plagiarism, modern code review, and mining software repositories. More info: https://cragkhit.github.io.

Jens Krinke is Associate Professor in the Software Systems Engineering Group at the University College London, where he is Director of CREST, the Centre for Research on Evolution, Search, and Testing. His main focus is software analysis for software engineering purposes. His current research interests include software similarity, modern code review, program analysis, and software testing. He is well known for his work on program slicing and clone detection.