SourcererCC: Scaling Code Clone Detection to Big-Code

Hitesh Sajnani[†] Vaibhav Saini[†] Jeffrey Svajlenko[‡]
Chanchal K. Roy[‡] Cristina V. Lopes[†]
[†]School of Information and Computer Science, UC Irvine, USA
{hsajani, vpsaini, lopes}@uci.edu
[‡]Department of Computer Science, University of Saskatchewan, Canada
{jeff.svajlenko, chanchal.roy}@usask.ca

ABSTRACT

Despite a decade of active research, there has been a marked lack in clone detection techniques that scale to large repositories for detecting near-miss clones. In this paper, we present a token-based clone detector, SourcererCC, that can detect both exact and near-miss clones from large inter-project repositories using a standard workstation. It exploits an optimized inverted-index to quickly query the potential clones of a given code block. Filtering heuristics based on token ordering are used to significantly reduce the size of the index, the number of code-block comparisons needed to detect the clones, as well as the number of required token-comparisons needed to judge a potential clone. We evaluate the scalability, execution time, recall and precision of SourcererCC, and compare it to four publicly available and state-of-the-art tools. To measure recall, we use two recent benchmarks: (1) a big benchmark of real clones, BigCloneBench, and (2) a Mutation/Injection-based framework of thousands of fine-grained artificial clones. We find SourcererCC has both high recall and precision, and is able to scale to a large inter-project repository (25K projects, 250MLOC) using a standard workstation.

1. INTRODUCTION

Clone detection locates exact or similar pieces of code, known as clones, within or between software systems. Clones are created when developers reuse code by copy, paste and modify, although clones may be created by a number of other means [28]. Developers need to detect and manage their clones in order to maintain software quality, detect and prevent new bugs, reduce development risks and costs, and so on [27, 28]. Clone management and clone research studies depend on quality tools. According to Rattan et al. [1], at least 70 diverse tools have been presented in the literature.

With the amount of source code increasing steadily, large-scale clone detection has become a necessity. Large-scale clone detection can be used for mining library candidates [16], detecting similar mobile applications [7], license violation detection [11, 21], reverse engineering product lines [11, 13], finding the provenance of a component [10], and code search [19, 20]. Large-scale clone detection allows researchers to study cloning in large software ecosystems (e.g., Debian), or study cloning in open-source development communities (e.g., GitHub). Developers often clone modules or fork projects to meet the needs of different clients, and need the help of large-scale clone detectors to merge these cloned systems towards a product-line style of development. These applications require tools that scale to hundreds of millions of lines of code. However, very few tools can scale to the demands of clone detection in very large code bases [27, 34].

A number of tools have been proposed to achieve a few specific applications of large-scale clone detection [7, 20, 21]. These tools make some assumptions regarding the requirements of their target domain that help with scalability. These domain-specific tools are not described as general large-scale clone detectors, and may face significant scalability challenges for general clone detection. General purpose clone detection is required for clone studies in large inter-project repositories and to help developers manage and merge their related software forks, as well as for use in the domain-specific activities. Scalable general purpose clone detection has been achieved by using deterministic [24] or non-deterministic [34] input partitioning and distributed execution of an existing non-scalable detector, using large distributed code indexes [15], or by comparing hashes after Type-1/2 normalization [16]. These existing techniques have a number of limitations. The novel scalable algorithms [15, 16] do not support Type-3 near-miss clones, where minor to significant editing activities might have taken place in the copy/pasted fragments, and therefore miss a large portion of the clones, since there are more Type-3 clones in the repositories than other types [27, 29, 33]. Type-3 clones can be the most needed in large-scale clone detection applications [7, 20, 27]. While input partitioning can scale existing non-scalable Type-3 detectors, this significantly increases the cumulative runtime, and requires distribution over a large cluster of machines to achieve scalability in absolute runtime [24, 34]. Distributable tools [24] can be costly and difficult to setup.

We set out to develop a clone detection technique and tool that would satisfy the following requirements: (1) accurate detection of near-miss clones, where minor to significant editing changes occur in the copy/pasted fragments; (2) programming language agnostic; (3) simple, non-distributed operation; and (4) scalability to hundreds of millions of lines of code. To that effect, we introduce SourcererCC, a token-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '16 Austin, Texas USA
Copyright 2016 ACM 978-1-4503-3900-1 ...$15.00.

based accurate near-miss clone detector that exploits an optimized index to scale to hundreds of millions of lines of code (MLOC) on a single machine. SourcererCC compares code blocks using a simple and fast bag-of-tokens[1] strategy which is resilient to Type-3 changes. Clone candidates of a code block are queried from a partial inverted index. A filtering heuristic is used to reduce the size of the index, which drastically reduces the number of required code block comparisons to detect the clones. It also exploits the ordering of tokens to measure a live upper and lower bound on the similarity of code blocks in order to reject or accept a clone candidate with fewer token comparisons. We found this technique has strong recall and precision for the first three clone types. SourcererCC is able to accurately detect exact and near-miss clones in 250MLOC on a single machine in only 4.5 days. We make two different versions of the tool available: (i) SourcererCC-B, a batch version of the tool that is more suitable for empirical analysis of the presence of clones in a system or a repository; and (ii) SourcererCC-I, an interactive version of the tool integrated with Eclipse IDE to help developers instantly find clones during software development and maintenance.

We evaluate the scalability, execution time and detection quality of SourcererCC. We execute it for inputs of various domains and sizes, including the large inter-project software repository IJaDataset-2.0 [3] (25,000 projects, 250MLOC, 3 million files), and observed good execution time and no scalability issues even on a standard machine with a 3.5GHz quad-core i7 CPU and 12GB of memory. We measure its clone recall using two proven [36, 37] clone benchmarks. We use BigCloneBench [33], a big benchmark of real clones that spans the four primary clone types and the full spectrum of syntactical similarity. We also use The Mutation and Injection Framework [26, 38], a synthetic benchmark that can precisely measure recall at a fine granularity. We measure precision by manually validating a sample of its output. We compare these results against publicly available popular and state-of-the-art tools, including CCFinderX [18], Deckard [17], iClones [12] and NiCad [9]. We find that SourcererCC is the only near-miss clone detector to scale to large repositories, and has the best execution time for very large inputs. SourcererCC also has strong precision and recall, and is competitive with the other tools.

The remainder of the paper is organized as follows. Section 2 describes important concepts and definitions. Section 3 presents SourcererCC's clone detection process in detail. Section 4 describes various experiments conducted to evaluate the scalability, recall and precision of SourcererCC against state-of-the-art tools on various benchmarks, with threats to validity discussed in Section 5. After drawing connections with the related work in Section 6, Section 7 concludes with a summary of the findings.

2. DEFINITIONS

The paper uses the following well-accepted definitions of code clones and clone types [5, 28]:

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

Clone Pair: A pair of code fragments that are similar, specified by the triple (f_1, f_2, ϕ), including the similar code fragments f_1 and f_2, and their clone type ϕ.

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

Code Block: A sequence of code statements within braces.

Type-1(T1): Identical code fragments, except for differences in white-space, layout and comments.

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

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

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

3. THE PROPOSED METHOD: SourcererCC

3.1 Problem Formulation

A software project P is represented as a set of code blocks $P : \{B_1, ..., B_n\}$. In turn, a code block B is represented as a bag-of-tokens (multiset) $B : \{T_1..., T_k\}$. A token is considered as programming language keywords, literals, and identifiers. A string literal is split on whitespace and operators are not included. Since a code block may have token multiplicity, each token is represented as a $(token, frequency)$ pair. Here, $frequency$ denotes the number of times $token$ appeared in a code block. This further reduces a code block representation to a set of $(token, frequency)$ pairs.

In order to quantitatively infer if two code blocks are clones, we use a similarity function which measures the degree of similarity between code blocks, and returns a non-negative value. The higher the value, the greater the similarity between the code blocks. As a result, code blocks with similarity value higher than the specified threshold are identified as clones.

Formally, given two projects P_x and P_y, a similarity function f, and a threshold θ, the aim is to find all the code block pairs (or groups) $P_x.B$ and $P_y.B$ $s.t$ $f(P_x.B, P_y.B) \geq \lceil \theta \cdot \max(|P_x.B|, |P_y.B|) \rceil$. Note that for intra-project similarity, P_x and P_y are the same. Similarly, all the clones in a project repository can be revealed by doing a self-join on the entire repository itself.

While there are many choices of similarity function, we use $Overlap$[3] because it intuitively captures the notion of overlap among code blocks. For example, given two code blocks B_x and B_y, the overlap similarity $OS(B_x, B_y)$ is computed as the number of tokens shared by B_x and B_y.

$$OS(B_x, B_y) = |B_x \cap B_y| \tag{1}$$

In other words, if θ is specified as 0.8, and $max(|B_x|, |B_y|)$ is t, then B_x and B_y should share at least $\lceil \theta |t| \rceil$ tokens to be identified as a clone pair. Note that if a token a appears in B_x twice and thrice in B_y, the match between B_x and B_y due to token a is two.

To detect all clone pairs in a project or a repository, the above approach of computing similarity between code blocks can simply be extended to iterate over all the code blocks

[1] Similar to the popular bag-of-words model [42] from Information Retrieval.

[3] The presented approach can be adapted for Jaccard and Cosine similarity functions as well.

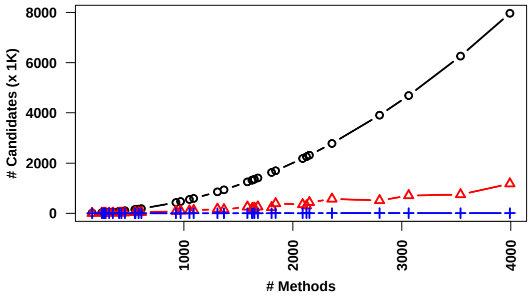

Figure 1: Growth in number of candidate comparisons with the increase in the number of code blocks

and compute pairwise similarity for each code block pair. For a given code block, all the other code blocks compared are called candidate code blocks or candidates in short.

While the approach is very simple and intuitive, it is also subjected to a fundamental problem that prohibits scalability - $O(n^2)$ time complexity. Figure 1 describes this by plotting the number of total code blocks (X-axis) vs. the number of candidate comparisons (Y-axis) in 35 Apache projects[2]. Note that the granularity of a code block is taken as a method. Points denoted by the ∘ show that the number of candidate comparisons increase quadratically[3] with the increase in number of methods. Later in Section 3 while describing SourcererCC, we will propose two filtering heuristics that significantly reduce the number of candidate comparisons during clone detection.

3.2 Overview

SourcererCC's general procedure is summarized in Figure 2. It operates in two primary stages: (i) partial index creation; and (ii) clone detection.

In the index creation phase, it parses the code blocks from the source files, and tokenizes them with a simple scanner that is aware of token and block semantics of a given language[4]. From the code blocks it builds an inverted index mapping tokens to the blocks that contains them. Unlike previous approaches, it does not create an index of all tokens in the code blocks, instead it uses a filtering heuristic (Section 3.3.1) to construct a partial index of only a subset of the tokens in each block.

In the detection phase, SourcererCC iterates through all of the code blocks and retrieves their candidate clone blocks from the index. As per the filtering heuristic, only the tokens within the sub-block are used to query the index, which reduces the number of candidate blocks. After candidates are retrieved, SourcererCC uses another filtering heuristic (Section 3.3.2), which exploits ordering of the tokens in a code block to measure a live upper-bound and lower-bound of similarity scores between the query and candidate blocks. Candidates whose upper-bound falls below the similarity threshold are eliminated immediately without further processing. Similarly, candidates are accepted as soon as their lower-bound exceeds the similarity threshold. This is repeated until the clones of every code block are located. SourcererCC exploits symmetry to avoid detecting the same clone twice. In the following sections, we provide a detailed description of the filtering heuristics and overall algorithm.

[2]The list is available at `http://mondego.ics.uci.edu/projects/SourcererCC/`.

[3]The curve can also be represented using $y = x(x-1)/2$ quadratic function where x is the number of methods in a project and y is the number of candidate comparisons carried out to detect all clone pairs.

[4]Currently we have support for Java, C and C# using TXL [8], but it can be easily extended to other languages.

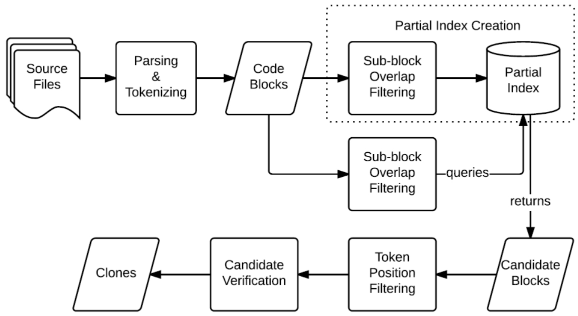

Figure 2: SourcererCC's clone detection process

3.3 Filtering Heuristics to Reduce Candidate Comparisons

3.3.1 Sub-block Overlap Filtering

The filtering heuristics are inspired by the work of Sarawagi et al. [32] and Vernica et al. [39] on set similarity. It follows an intuition that when two sets have a large overlap, even their smaller subsets overlap. Since we represent code blocks as bag-of-tokens (i.e., a multiset), we can extend this idea to code blocks, i.e., when two code blocks have large overlap, even their smaller sub-blocks should overlap as shown in [31]. Formally, we state it as the following property:

Property 1: *Given blocks B_x and B_y consisting of t tokens each in some predefined order, if $|B_x \cap B_y| \geq i$, then the sub-blocks SB_x and SB_y of B_x and B_y respectively, consisting of first $t - i + 1$ tokens, must have at least one matching token.*

To understand the implications of this property in clone detection, let us consider two code blocks $B_x = \{\mathbf{a}, \mathbf{b}, c, d, e\}$ and $B_y = \{\mathbf{b}, \mathbf{c}, d, e, f\}$ with 5 tokens ($t = 5$) each. Let θ be specified as 0.8 meaning that the two blocks should match at least $\lceil 0.8 * 5 \rceil = 4$ tokens to be considered clones i.e, ($i = 4$).

According to Property 1, in order to find out if B_x and B_y are clones, we only need to check if their sub-blocks consisting of first $t - i + 1 = 2$ tokens match at least one token. In this case, they do, as token b is common in both the sub-blocks (marked in bold). However, if they had not shared any token, then even without looking at the remaining tokens of the blocks, we could have most certainly figured that B_x and B_y will not end up as a clone pair for the given θ. In other words, Property 1 suggests that instead of comparing all the tokens of B_x and B_y against each other, we could compare only their sub-blocks consisting of first $t - i + 1$ tokens to deduce if B_1 and B_2 will *not* be clones.

In order to apply Property 1, the tokens in the code blocks need to follow a predefined global order. While there are many ways in which tokens in a block can be ordered e.g., alphabetical order, length of tokens, occurrence frequency of token in a corpus, etc., a natural question is what order is most effective in this context. As it turns out, software vocabulary exhibits very similar characteristics to natural languages corpus and also follow Zipf's law [14, 43]. That is, there are few very popular (frequent) tokens, and the frequency of tokens decreases very rapidly with rank. In other words, while most of the code blocks are likely to contain one or more of few very popular tokens (e.g., keywords, or common identifier names like i, j, *count*, etc.) not many will share rare tokens (e.g., identifiers that are domain or project specific). So if code blocks are ordered according to the popularity of tokens in the corpus, naturally, their sub-blocks will consist of these rare tokens. Such arrangement will ensure low probability of different sub-blocks sharing similar token. In other words, this ordering will eliminate more false positive candidates[5].

To describe how effective this filtering is, points denoted by △ in Figure 1 show the number of candidate comparisons after applying the filtering. The difference with the earlier curve (○) show the impact of filtering in eliminating candidate comparisons.

The below section discusses when the use of Property 1 may still be ineffective and demonstrate how ordering of tokens in a code block can be further exploited to formalize yet another filtering heuristic that is extremely effective in eliminating even more candidate comparisons.

3.3.2 Token Position Filtering

In order to understand when Property 1 may be ineffective, consider code blocks B_x and B_y from the previous example, except B_x now has one fewer token. Hence $B_x = \{\mathbf{a},\, \mathbf{b}, c, d\}$ and $B_y = \{\mathbf{b},\, \mathbf{c}, d, e, f\}$.

Assuming the same value of θ, the blocks must still match tokens ($\lceil \theta \cdot \max(|B_x|, |B_y|) \rceil = \lceil 0.8 * 5 \rceil = 4$) to be a clone pair. But since the two blocks have only 3 tokens in common, they cannot be identified as a clone pair. However, note that their sub-blocks (shown in bold) consisting of first $t-i+1 = 2$ tokens still have a common token b. As a result, *Property 1* is satisfied and B_y will be identified as a candidate of B_x although B_x and B_y eventually will not end up as a clone pair. In general, cases when the code blocks have fairly different sizes it is likely that they may result in false positives (and rejected) even after satisfying Property 1.

Interestingly, to overcome this limitation, the ordering of tokens in code blocks can be exploited. For example, if we closely examine the position of the matched token b in B_x and B_y, we can obtain an estimate of the maximum possible overlap between B_x and B_y as the sum of *current matched tokens* and the *minimum number of unseen tokens in* B_x *and* B_y, i.e., $1+min(2,4) = 3$. Since this upper bound on overlap is already smaller than the needed threshold of 4 tokens, we can safely reject B_y as a candidate of B_x. Note that we can compute a safe upper bound (without violating the correctness) because the tokens follow a predefined order. The above heuristic can be formally stated as follows.

Property 2: *Let blocks* B_x *and* B_y *be ordered and* $\exists$ *token* t *at index* i *in* B_x, *s.t* B_x *is divided in to two parts, where* $B_x(first) = B_x[1...(i-1)]$ *and* $B_x(second) = B_x[i...|B_x|)]$.

Now if $|B_x \cap B_y| \geq \lceil \theta \cdot \max(|B_x|, |B_y|) \rceil$, *then* $\forall\, t \in B_x \cap B_y$, $|B_x(first) \cap B_y(first)| + min(|B_x(second)|, |B_y(second)|) \geq \lceil \theta \cdot \max(|B_x|, |B_y|) \rceil$.

To describe how effective this filtering is, points denoted by + in Figure 1 show the number of candidate comparisons after applying this filtering. The reduction is so significant that empirically on this dataset, the function seems to be *near-linear*. This is a massive reduction in comparison with the quadratic function shown earlier without any filtering.

Although both the filtering heuristics are independent of each other, they complement each other to effectively reduce more number of candidate comparisons together than alone. The index data structure in conjunction with the above filtering heuristics form the key components of SourcererCC to achieve scalability. The next section describes the complete algorithm of SourcererCC.

3.4 Clone Detection Algorithm

The algorithm works in two stages: (i) Partial Index Creation; and (ii) Clone Detection. Each step has filtering heuristics directly embedded in it as described below.

Partial Index Creation. In traditional index-based approaches, all the tokens are indexed. However, SourcererCC's index creation step exploits Property 1 and creates indexes for tokens only in sub-blocks. We call this *Partial Index*. This not only saves space but also enables faster retrieval because of a smaller index.

Algorithm 1 lists the steps to create a partial index. The first step is to iterate over each code block b (*line* 3), and sort it according to the global token frequency map (GTP) (*line* 4). This is done as a pre-requisite to the application of filtering based on Property 1. Next, the size of sub-block is computed using formula shown in Property 1 i.e., $(t-i+1)$. Later, tokens in the sub-block are indexed to create the partial index (*lines* $6-8$).

Algorithm 1 SourcererCC's Algorithm - Partial Index Creation

INPUT: B is a list of code blocks $\{b_1,\, b_2, ...b_n\}$ in a project/repository, GTP is the global token position map, and θ is the similarity threshold specified by the user
OUTPUT: Partial Index(I) of B

```
1:  function createPartialIndex(B, θ)
2:      I = φ
3:      for each code block b in B do
4:          b = Sort(b, GTP)
5:          tokensToBeIndexed = |b| − ⌈θ · |b|⌉ + 1
6:          for i = 1 : tokensToBeIndexed do
7:              t = b[i]
8:              I_t = I_t ∪ (t, i)
9:          end for
10:     end for
11:     return I
12: end function
```

Clone Detection. After the partial index is created, the goal is to detect clones. Algorithm 2 describes the steps in detail. The *detectClones()* function iterates over each query block b, and sorts them using the same (GTP) that was created during index creation (*line* 4). Again, this is done as a prerequisite for both Property 1 & 2 to be applicable. After that, it calculates the length of query sub-block by using the same formula described in Property 1 (*line* 5). Next it iterates over only as many tokens as the length of b's sub-block and retrieves candidates by querying the partial index. Note that since the partial index is created using only sub-blocks, the candidates retrieved in this phase implicitly satisfy Property 1. In other words, by creating the partial index, the algorithm not only reduces the index size, but also ensures that we only get a filtered set of candidates that satisfy Property 1.

After the candidates are retrieved for a given query block, a trivial optimization to further eliminate candidates is done using size of the candidates. That is, if a candidate c does not have enough tokens needed for it to be b's clone pair,

[5] Candidates that eventually will not be identified as clones of a code block are known as false positive candidates for that code block.

then there is no point in even comparing them. This is done using a conditional check $|c| > \lceil \theta \cdot |b| \rceil$ on *line* 8. This further filters out false positive candidates.

The remaining candidates that have satisfied the above elimination process are now subjected to the filtering based on Property 2. First, based on θ, a threshold is computed that identifies the minimum number of tokens needed to be matched for b and c to be identified as a clone pair (*ct* on *line* 9). Now, as the tokens in b and c are compared, a theoritical upper bound is dynamically computed based on the number of remaining tokens in b and c (*line* 10). This upper bound indicates the maximum number of tokens b and c could match assuming all of their tokens will match. If at any point in the iteration, the sum of upper bound (i.e, maximum number of tokens b and c could match) and the current similarity score (i.e, number of tokens b and c have matched) happens to be less than *ct* (i.e, minimum number of tokens b and c need to match), c is eliminated from b's candidate map *candSimMap* (*lines* 11 *and* 14). In other words, it is violation of Property 2. On the other hand, if

Algorithm 2 SourcererCC's Algorithm - Clone Detection

INPUT: B is a list of code blocks $\{b_1, b_2,...b_n\}$ in a project/repository, I is the partial index created from B, and θ is the similarity threshold specified by the user
OUTPUT: All clone classes (*cloneMap*)

```
1:  function detectClones(B, I, θ)
2:    for each code block b in B do
3:      candSimMap = φ
4:      b = Sort(b, GTP)
5:      querySubBlock = |b| − ⌈ θ ·|b|⌉ + 1
6:      for i = 1 : querySubBlock do
7:        t = b[i]
8:        for each (c, j) ∈ I_t such that |c| > ⌈ θ · |b| ⌉ do
9:          ct = ⌈ max(|c|, |b|) · θ ⌉
10:         uBound = 1 + min(|b| − i, |c| − j)
11:         if candSimMap[c] + uBound ≥ ct then
12:           candSimMap[c] = candSimMap[c] + (1, j)
13:         else
14:           candSimMap[c] = (0, 0)              ▷ eliminate c
15:         end if
16:       end for
17:     end for
18:     verifyCandidates(b, candSimMap, ct)
19:   end for
20:   return cloneMap
21: end function
```

```
1:  function verifyCandidates(b, candSimMap, ct)
2:    for each c ∈ candSimMap such that candSimMap[c] > 0 do
3:      tokPos_c = Position of last token seen in c
4:      tokPos_b = Position of last token seen in b
5:      while tokPos_b < |b| && tokPos_c < |c| do
6:        if min(|b| − tokPos_b, |c| − tokPos_c) ≥ ct then
7:          if b[tokPos_b] == c[tokPos_c] then
8:            candSimMap[c] = candSimMap[c] + 1
9:          else
10:           if GTP[b[tokPos_b]] < GTP[c[tokPos_c]] then
11:             tokPos_b + +
12:           else
13:             tokPos_c + +
14:           end if
15:         end if
16:       else
17:         break
18:       end if
19:     end while
20:     if candSimMap[c] > ct then
21:       cloneMap[b] = cloneMap[b] ∪ c
22:     end if
23:   end for
24: end function
```

the sum is more than *ct*, the similarity between b and c gets updated with each token that is matched (*line* 12). Once all the tokens in b's sub-block are exhausted (*line* 19), we have a map of candidates (*candSimMap*) along with their similarity score and the last seen token in each candidate. The reason for storing the last seen token will become clear as we explain futher. The next task is to verify if the candidates will eventually end up being b's clones. This is done in a call to *verifyCandidates()* function on *line* 18.

Candidate Verification. The goal of *verifyCandidates()* function is to iterate over candidates c of query b that were not rejected in *detectClones()*, compute their similarity score with b, and reject them if the score does not meet the computed threshold *ct*) or add them to the *cloneMap* if it does.

In doing so, an important optimization is seen on (*line* 5). Note that tokens are not iterated from the start but from last token seen in b and c because earlier in *detectClones()* few tokens of b and c were already iterated to check if they satisfy Property 1 & 2 (*lines* $6-8$). Hence the function avoids iterating over those tokens again. It is for this reason, in *detectClones()*, *candSimMap* is designed to not only store candidates but also the last token that seen in each candidate, i.e., ($Candidate, TokensSeenInCandidate$) pair.

The rest of the function while iterating over the remaining tokens ensures that Property 2 holds at every iteration (*line* 6), and then increments the similarity score whenever there is a token match (*lines* $7-8$). If at any iteration, Property 2 is violated, candidate is eliminated immediately without iterating over the remaining tokens (*line* 17). Thus saving much computation.

Another trivial but important optimization is done while iterating over code blocks. Since b and c are already sorted using a global token frequency (GTP), *verifyCandidates()* efficienty iterates over b and c by incrementing only the index of a block that has a lower globally ranked token (*lines* $10-14$). Hence while iterating, except in the worst case when b & c happen to be clone pairs, time complexity is reduced from $O(|b| * |c|)$ to $O(|b| + |c|)$.

3.5 Detection of Near-miss (Type-3) clones

One of the distinguishing characteristics of SourcererCC compared to other token-based tools is its ability to detect Near-miss (Type-3) clones. The bag-of-tokens model plays an important role in this. Type-3 clones are created by adding, removing or modifying statements in a duplicated code fragment. Since the bag-of-tokens model is agnostic to relative token positions in the code block, it is resilient to such changes, and hence can detect near-miss clones as long as the code blocks (bags) share enough tokens to exceed a given overlap threshold.

Many Type-3 clones have modifications such as swapping statement positions in code blocks, combining multiple condition expressions into one, changing operators in conditional statements, and use of one language construct over another (for vs while). While these changes may exhibit semantic difference, they preserve enough syntactic similarity at a token level to be detected as similar. Detecting such clones can be difficult for other token-based approaches as they use token sequences as a unit of match [18]. While a token-sequence approach could merge nearby cloned sequences into Type-3 clones [12], they fail to detect the clones when the Type-3 gaps are too frequent or large.

Table 1: Clone Detection Tool Configurations

Tool	Scale/BigCloneBench	Mutation Framework
SourcererCC	Min length 6 lines, min similarity 70%, function granularity.	Min length 15 lines, min similarity 70%, function granularity.
CCFinderX	Min length 50 tokens, min token types 12.	Min length 50 tokens, min token types 12.
Deckard	Min length 50 tokens, 85% similarity, 2 token stride.	Min length 100 tokens, 85% similarity, 4 token stride.
iClones	Min length 50 tokens, min block 20 tokens.	Min length 100 tokens, min block 20 tokens.
NiCad	Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity.	Min length 15 lines, blind identifier normalization, identifier abstraction, min 70% similarity.

4. EVALUATION

In this section, we evaluate the execution and detection performance of SourcererCC. We begin by evaluating its execution time and scalability using subject inputs of varying sizes in terms of lines of code (LOC). We then demonstrate SourcererCC's execution for a large inter-project repository, one of the prime targets of scalable clone detection. We measure its clone recall using two benchmarks: The Mutation and Injection Framework [26, 38] and BigCloneBench [33, 37]. We measure precision by manually validating a sample of its output for the BigCloneBench experiment.

We compare SourcererCC's execution and detection performance against four publicly available clone detection tools, including: CCFinderX [18], Deckard [17], iClones [12] and NiCad [9]. We include CCFinderX as it is a popular and successful tool, which has been used in many clone studies. We include Deckard, iClones and NiCad as popular examples of modern clone detection tools that support Type-3 clone detection. While we have benchmarked a number of tools in our previous work [36, 37], we focus on those with the best scalability, recall, and/or most unique performance aspects for this study. We focus primarily on near-miss clone detectors, as Type-1 and Type-2 clones are relatively easy to detect. The configurations of these tools for the experiments are found in Table 1. These are targeted configurations for the benchmarks, and are based on our extensive previous experiences [36, 37] with the tools, as well as our previous discussions with their developers, where available.

Our primary goal with SourcererCC is to provide a clone detection tool that scales efficiently for large inter-project repositories with near-miss (Type-3) clone detection capability. Most existing state-of-the-art tools have difficulty with such large inputs, and fail due to scalability limits [34, 35]. Common limits include untenable execution time, insufficient system memory, limitations in internal data-structures, unexplained crashing, or reporting an error due to their design not anticipating such a large input [34, 35]. We consider SourcererCC successful if it can scale to a large inter-project repository without encountering these scalability constraints while maintaining a clone recall and detection precision comparable to the state-of-the-art. As our target we use IJaDataset 2.0 [3], a large inter-project Java repository containing 25,000 open-source projects (3 million source files, 250MLOC) mined from SourceForge and Google Code.

4.1 Execution Time and Scalability

In this section, we evaluate the execution time and scalability of SourcererCC and compare it to the competing tools. Execution time primarily scales with the size of the input in terms of the number of lines of code (LOC) needed to be processed and searched by the tool. So this is the ideal input property to vary while evaluating execution performance and scalability. However, it is difficult to find subject systems that are large enough and conveniently dispersed in size. Additionally, a tool's execution time and memory requirements may also be dependent on the clone density, or other properties of the subject systems. It is difficult to control for these factors while measuring execution performance and scalability in terms of input size.

Our solution was to build inputs of varying convenient sizes by randomly selecting files from IJaDataset. This should ensure each input has similar clone density, and other properties that may affect execution time, except for the varying size in LOC. Each input has the properties of an inter-project repository, which is a target of large-scale clone detection. We created one input per order of magnitude from 1KLOC to 100MLOC. We built the inputs such that each larger input contains the files of the smaller inputs. This ensures that each larger subset is a progression in terms of execution requirements. Lines of code was measured using the unix tool 'cloc' [2], and includes only lines containing code, not comment or blank lines.

The execution time of the tools for these inputs can be found in Table 2. The tools were executed for these inputs using the configurations listed under "Scale" in Table 1. The tools were executed on a machine with a 3.5GHz quad-core i7 CPU, 12GB of memory, and a 250GB solid-state drive. We use a 12GB configuration to approximate the average workstation where 8GB and 16GB are standard. While the tools may perform better on 32GB+ configurations, this is not typical of the average workstation. We limit the tools to 10GB to account for OS memory usage and to prevent paging. We use the same configurations for evaluating recall with BigCloneBench such that recall, execution performance and scalability can be directly compared.

Scalability. SourcererCC is able to scale even to the largest input with reasonable execution time given the input sizes. CCFinderX is the only competing tool to scale to 100MLOC, however it only detects Type-1 and Type-2 clones. The competing Type-3 tools encounter scalability limits before the 100MLOC input. Deckard and iClones run out of memory at the 100MLOC and 1MLOC inputs, respectively. NiCad is able to scale to the 10MLOC input, but refuses to execute clone detection on the 100MLOC input. In our previous experience [35], NiCad refuses to run on inputs that exceeds its internal data-structure limits, which prevent executions that will take too long to complete. From our experiment, it is clear that the state-of-the-art Type-3 tools do not scale to large inputs, whereas SourcererCC can.

Execution Time. For the 1KLOC to 100KLOC inputs, SourcererCC has comparable execution time to the competing tools. iClones is the fastest, but it hits scalability issues (memory) as soon as the 1MLOC input. SourcererCC has comparable execution time to CCFinderX and NiCad for the 1MLOC input, but is much faster than Deckard. SourcererCC has comparable execution time to CCFinderX for the 10MLOC input size, but is much faster than NiCad. For the largest input size, SourcererCC is twice as fast as CCFinderX, although their execution times fall within the same order of magnitude. Before the 100MLOC input, SourcererCC and CCFinderX have comparable execution times.

Table 2: Execution Time (or Failure Condition) for Varying Input Size

LOC	SourcererCC	CCFinderX	Deckard	iClones	NiCad
1K	3s	3s	2s	**1s**	1s
10K	6s	4s	9s	**1s**	4s
100K	15s	21s	1m 34s	**2s**	21s
1M	**1m 30s**	2m 18s	1hr 12m 3s	MEMORY	4m 1s
10M	32m 11s	**28m 51s**	MEMORY	—	11hr 42m 47s
100M	**1d 12h 54m 5s**	3d 5hr 49m 11s	—	—	INTERNAL LIMIT

SourcererCC is able to scale to inputs of at least 100MLOC. Its execution time is comparable or better than the competing tools. Of the examined tools, it is the only state-of-the-art Type-3 clone detector able to scale to 100MLOC. While CCFinderX can scale to 100MLOC for only detecting Type-1 and Type-2 clones, SourcererCC completes in half the execution time while also detecting Type-3 clones.

4.2 Experiment with IJaDataset

Since SourcererCC scaled to 100MLOC without issue, we also executed it for the entire IJaDataset (250MLOC). This represents the real use case of clone detection in a large inter-project software repository. We execute the tool on a standard workstation with a quad-core i7 CPU, 12GB of memory and solid state drive. We restricted the tool to 10GB of memory and 100GB of SSD disk space. We executed SourcererCC using the "Scale" configuration in Table 1, with the exception of increasing the minimum clone size to ten lines. Six lines is common in recall benchmarking [5]. However, a six line minimum may cause an excessive number of clones to be detected in IJaDataset, and processing these clones for a research task can become another difficult scalability challenge [34]. Additionally, larger clones may be more interesting since they capture a larger piece of logic, while smaller clones may be more spurious.

SourcererCC successfully completed its execution for IJaDataset in 4 days and 12 hours, detecting a total of 146 million clone pairs. The majority of this time was clone detection. Extracting and tokenizing the functions required 3.5 hours, while computing the global token freqeuncy map and tokenizing the blocks required only 20 minutes. SourcererCC required 8GB of disk space for its pre-processing, index (1.2GB) and output. Of the 4.7 million functions in IJaDataset greater than 10 lines in length, 2.4 million (51%) appeared in at least one clone pair detected by SourcererCC. We have demonstrated that SourcererCC scales to large inter-project repositories on a single machine with good execution time. We have also shown that building an index is an inexpensive way to scale clone detection and reduce overall execution time.

Since CCFinderX scales to the 100MLOC sample, we also executed it for IJaDataset. We used the same settings as the scalability experiment. We did not increase CCFinderX's minimum clone size from 50 tokens, which is roughly 10 lines (assuming 5 tokens per line). CCFinderX executed for 2 days before crashing due to insufficient disk space. Its pre-processed source files (25GB) and temporarily disk space usage (65GB) exceeded the 100GB reserved space. Based on the findings of a previous study, where CCFinder was distributed over a cluster of computers [24], we can estimate it would require tens of days to complete detection on 250MLOC, given sufficiently large disk-space. So we can confidently say that SourcererCC is able to complete sooner, while also detecting Type-3 clones.

4.3 Recall

In this section we measure the recall of SourcererCC and the competing tools. Recall has been very difficult for tool developers to measure as it requires knowledge of the clones that exist in a software system [27, 28]. Manually inspecting a system for clones is non-trivial. Even a small system like Cook, when considering only function clones, has almost a million function pairs to inspect [40]. Bellon et al. [5] created a benchmark by validating clones reported by the clone detectors themselves. This has been shown to be unreliable for modern clone detectors [36]. Updating this benchmark to evaluate a modern tool would require extensive manual clone validation with a number of modern tools. As such, many clone detection tool papers simply do not report recall.

In response, we created The Mutation and Injection Framework [26, 38], a synthetic benchmark that evaluates a tool's recall for thousands of fine-grained artificial clones in a mutation-analysis procedure. The framework is fully automatic, and requires no validation efforts by the tool developer. However, we recognized that a modern benchmark of real clones is also required. So we developed an efficient clone validation strategy based on code functionality and built BigCloneBench [33], a big clone benchmark containing 8 million validated clones within and between 25,000 open-source projects. It measures recall for an extensive variety of real clones produced by real developers. The benchmark was designed to support the emerging large-scale clone detection tools, which previously lacked a benchmark. This combination of real-world and synthetic benchmarking provides a comprehensive view of SourcererCC's clone recall.

4.3.1 *Recall Measured by The Mutation Framework*

The Mutation Framework evaluates recall using a standard mutation-analysis procedure. It starts with a randomly selected real code fragment (a function or a code block). It mutates this code fragment using one of fifteen clone-producing mutation operators. Each mutation operator performs a single code edit corresponding to one of the first three clone types, and are based on an empirically validated taxonomy of the types of edits developers make on copy/pasted code. This artificial clone is randomly injected into a copy of a subject system. The clone detector is executed for this system, and its recall measured for only the injected clone. The framework requires the tool to not only sufficiently report the injected clone, but also appropriately handle the clone-type specific change(s) introduced by the mutation. As per mutation-analysis, this is repeated thousands of times. Further details, including the list of mutation operators, are available in our earlier studies [26, 30, 38].

Procedure. We executed the framework for Java, C and C# clones using the following configuration. For each language, we set the framework to generate clones using 250 randomly selected functions, 10 randomly selected injection locations, and the 15 mutation operators, for a total

Table 3: Mutation Framework Recall Results

Tool	Java			C			C#		
	T1	T2	T3	T1	T2	T3	T1	T2	T3
SourcererCC	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
CCFinderX	99	70	0	**100**	77	0	**100**	78	0
Deckard	39	39	37	73	72	69	-	-	-
iClones	**100**	92	96	99	96	99	-	-	-
NiCad	**100**	**100**	**100**	99	99	99	98	98	98

of 37,500 unique clones per language (112,500 total). For Java we used JDK6 and Apache Commons as our source repository and IPScanner as our subject system. For C we used the Linux Kernel as our repository and Monit as our subject system. For C# we use Mono and MonoDevelop as our repository, and MonoOSC as our subject system. We constrained the synthesized clones to the following properties: (1) 15-200 lines in length, (2) 100-2000 tokens in length, and (3) a mutation containment of 15%. We have found this configuration provides accurate recall measurement [36, 37]. The tools were executed and evaluated automatically by the framework using the configurations listed in Table 1. To successfully detect a reference (injected) clone, a tool must report a candidate clone that subsumes 70% of the reference clone by line, and appropriately handle the clone-type specific edit introduced by the mutation operator [38].

Results. Recall measured by the Mutation Framework for SourcererCC and the competing tools is summarized in Table 3. Due to space considerations, we do not show recall per mutation operator. Instead we summarize recall per clone type. SourcererCC has perfect recall for the first three clone types, including the most difficult Type-3 clones, for Java, C and C#. This tells us that its clone detection algorithm is capable of handling all the types of edits developers make on copy and pasted code for these languages, as outlined in the editing taxonomy for cloning [30].

SourcererCC exceeds the competing tools with the Mutation Framework. The runner up is NiCad, which has perfect recall for Java, and near-perfect recall for C and C#. iClones is also competitive with SourcererCC, although iClones has some troubles with a small number of Type-2 and Type-3 clones. SourcererCC performs much better for Type-2 and Type-3 clones than CCFinderX. Of course, as a Type-2 tool, CCFinderX does not support Type-3 detection. SourcererCC performs much better then Deckard across all clone types. While Deckard has decent recall for the C clones, its Java recall is very poor. We believe this is due to its older Java parser (Java-1.4 only), while the Java reference clones may contain up to Java-1.6 features.

SourcererCC has perfect recall with the Mutation Framework, which shows it can handle all the types of edits developers make on cloned code. As per standard mutation analysis, the Mutation Framework only uses one mutation operator per clone. This allows it to measure recall very precisely per type of edit and clone type. It also prevents the code from diverging too far from natural programming. However, this means that the framework makes simple clones. It does not produce complex clones with multiple type of edits, and the Type-3 clones it produces generally have a higher degree of syntactical similarity. To overcome this issue, we use the real-world benchmark BigCloneBench as follows.

4.3.2 *Recall Measured by BigCloneBench*

Here we measure the recall of SourcererCC using BigCloneBench and compare it to the competing tools. We evaluate how its capabilities shown by the Mutation Framework translate to recall for real clones produced by real developers in real software-systems, spanning the entire range of clone types and syntactical similarity. Together the benchmarks provide a complete view of SourcererCC's recall.

Table 4: BigCloneBench Clone Summary

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

BigCloneBench [33] is a big clone benchmark of manually validated clone pairs in the inter-project software repository IJaDataset 2.0 [3]. IJaDataset consists of 25,000 open-source Java systems spanning 3 million files and 250MLOC. BigCloneBench was built by mining IJaDataset for functions implementing particular functionalities. Each clone pair is semantically similar (by their target functionality) and is one of the four primary clone types (by their syntactical similarity). The published version of the benchmark considers 10 target functionalities [33]. We use an in-progress snapshot of the benchmark with 43 target functionalities, and 8 million validated clone pairs, for this study.

For this experiment, we consider all clones in BigCloneBench that are 6 lines and 50 tokens in length or greater. This is the standard minimum clone size for measuring recall [5, 37]. By specifying this both in lines and tokens we are able to configure the tools appropriately for clone size (Table 1). Clone size is a primary clone detection configuration, and this prevents it from biasing the comparison of the tools' recall. The number of clones in BigCloneBench, given this size constraint, is summarized per clone type in Table 4. There is no agreement on when a clone is no longer syntactically similar, so it is difficult to separate the Type-3 and Type-4 clones in BigCloneBench. Instead we divide the Type-3 and Type-4 clones into four categories based on their syntactical similarity, as follows. Very Strongly Type-3 (VST3) clones have a syntactical similarity between 90% (inclusive) and 100% (exclusive), Strongly Type-3 (ST3) in 70-90%, Moderately Type-3 (MT3) in 50-70% and Weakly Type-3/Type-4 (WT3/4) in 0-50%. Syntactical similarity is measured by line and by token after Type-1 and Type-2 normalizations. We use the smaller of the measurements for categorization. The categories, and the benchmark in general, are explained in more detail elsewhere [33].

Procedure. We executed the tools for IJaDataset and evaluated their recall with BigCloneBench. As we saw previously (Section 4.1), most tools do not scale to the order of magnitude of IJaDataset (250MLOC). Our goal here is to measure recall not scalability. We avoid the scalability issue by executing the tools for a reduction of IJaDataset with only those files containing the known true and false clones in BigCloneBench (50,532 files, 10MLOC). Some of the competing tools have difficulty even with the reduction, in which case we partition it into small sets, and execute the tool for each pair of partitions. In either case, the tool is exposed to every reference clone in BigCloneBench, and it is also exposed to a number of false positives as well, creating a realistic input. We measure recall using a subsume-based clone-matching algorithm with a 70% threshold. A tool successfully detects a reference clone if it reports a candidate clone that subsumes 70% of the reference clone by line. This is the same algorithm we use with the Mutation Framework, and is a standard in benchmarking [5].

Results. Recall measured by BigCloneBench is summarized in Table 5. It is is summarized per clone type and per Type-3/4 category for all clones, as well as specifically for the intra and inter-project clones.

SourcererCC has perfect detection of the Type-1 clones in BigCloneBench. It has near-perfect Type-2 detection, with negligible difference between intra and inter-project. This shows that the 70% threshold is sufficient to detect the Type-2 clones without identifier normalizations. SourcererCC has excellent Type-3 recall for the VST3 category, both in the general case (93%) and for intra-project clones (99%). The VST3 recall is still good for the inter-project clones (86%), but it is a little weaker. SourcererCC's Type-3 recall begins to drop off for the ST3 recall (61%). Its recall is good in this Type-3 category for the intra-project clones (86%) but poor for the inter-project clones (48%). We believe this is due to inter-project Type-3 clones having a higher incidence of Type-2 differences, causing them to not exceed SourcererCC's 70% overlap threshold. Remember that the reference clone categorization is done using syntactical similarity measured after Type-2 normalizations, whereas SourcererCC does not normalize the identifier token names (to maintain precision and index efficiency). Lowering SourcererCC's threshold would allow these to be detected, but could harm precision. SourcererCC has poor recall for the MT3 and WT3/T4, which is expected as these clones fall outside the range of syntactical clone detectors [37]. Type-4 detection is outside the scope of this study.

Compared to the competing tools, SourcererCC has the second best recall overall, with NiCad taking the lead. Both tools have perfect Type-1 recall, and they have similar Type-2 recall, with NiCad taking a small lead. SourcererCC has competitive VST3 recall, but loses out in the inter-project case to NiCad. SourcererCC is competitive with NiCad for intra-project clones in the ST3 category, but falls significantly behind for the inter-project case and overall. NiCad owes its exceptional Type-3 recall to its powerful source normalization capabilities. However, as we saw previously in Section 4.1, NiCad has much poorer execution time for larger inputs, and hits scalability constrains at the 100MLOC input. So SourcererCC instead competes with execution performance and scalability.

Comparison to CCFinderX is interesting as it is the only other tool to scale to the 100MLOC input. Both tools have comparable Type-1 and Type-2 recall, with SourcererCC having the advantage of also detecting Type-3 clones, the most difficult type. While BigCloneBench is measuring a non-negligible VST3 recall for CCFinderX, it is not truly detecting the Type-3 clones. As shown by the Mutation Framework in Table 3, CCFinderX has no recall for clones with Type-3 edits, while SourcererCC has perfect recall. Rather, CCFinderX is detecting significant Type-1/2 regions in these (very-strongly similar) Type-3 clones that satisfy the 70% coverage threshold. This is a known limitation in real-world benchmarking [36, 37], which is why both real-world and synthetic benchmarking is needed. CCFinderX's detection of these regions in the VST3 is not as useful to users as they need to manually recognize the missing Type-3 features. CCFinderX's Type-3 recall drops off past the VST3 category, where Type-3 gaps are more frequent in the clones. While we showed previously that CCFinderX also scales to larger inputs (Section 4.1), SourcererCC's faster execution, Type-3 support and better recall make it an ideal choice for large-scale clone detection.

Deckard and iClones are the other competing Type-3 clone detectors. Both SourcererCC and iClones have perfect Type-1 recall, but SourcererCC exceeds iClones in both Type-2 and Type-3 detection, and iClones does not scale well. Deckard has poor overall recall for all clone types, along with its scalability issues.

4.4 Precision

Unlike clone detection recall, where there exists high-quality benchmarks [38, 33], measuring precision remains an open problem, and there is no standard benchmark or methodology. Instead, we estimate the precision of the tools by manually validating a random sample of their output, which is the typical approach. From each tool we randomly selected 400 of the clone pairs they detected in the recall experiment. The validation efforts were equally distributed over five judges, all software researchers, with each validating 80 clones from each tool. The clones were shuffled and the judges were kept blind of the source of each clone. The judges were familiar with the cloning definitions, and were asked to validate the clones as per their judgment.

We find that SourcererCC has a precision of 83%, the second best precision of these tools. This is a very strong precision as per the literature [27, 30, 28], and demonstrates the accuracy and trustworthiness of SourcererCC's output. We summarize the precision of all the tools in Table 6, and contrast it against their overall and Type-3 recall measured by BigCloneBench. We do not include the MT3 and WT3/T4 clones as they are outside the scope of these tools. iClones has the top precision (91%) because it is cautious when reporting Type-3 clones, although this results in a Type-3 recall (38%) significantly below SourcererCC (68%) and NiCad (96%). SourcererCC's bag-of-tokens model and similarity threshold allows it to provide a good balance of recall and precision, achieving the 2nd best Type-3 recall, while also providing superior scalability. NiCad has a precision of 56%, possibly because of its use of normalizations and relaxed threshold. However, with these settings NiCad has a very strong overall (99%) and Type-3 (96%) recall and among the top of the tools. The authors [26, 38] report a precision of 89-96% for NiCad, depending on the configurations. CCFinderX's precision, while competitive, is low considering it only targets Type-1 and Type-2 clones (although it detects some Type-1/2 regions in Type-3 clones). Deckard has very poor precision in this experiment, reporting some clones that are very dissimilar. This may be because we relaxed its similarity threshold to detect more Type-3 clones. The authors [17] report a precision of 94% for Java-1.4 code with a 100% similarity threshold. Nonetheless, CCFinderX and Deckard show very poor Type-3 recall as well.

Tool configuration, particularly minimum clone size, is a bias in this precision experiment. This was controlled in the recall experiment by setting a minimum clone size of six lines and 50 tokens in BigCloneBench, and configuring the tools appropriately. However, there is no agreement between lines of code and the tokens contained, and even the tools measure lines (original/pretty-printed) and tokens (original/filtered) in different ways. This makes comparing the precision of the tools difficult because this configuration issue may cause a tool to detect many small spurious clones that another tool does not due to difference in clone size configuration and/or measurement. To examine this, we re-measured precision

Table 5: BigCloneBench Recall Measurements

Tool	All Clones						Intra-Project Clones						Inter-Project Clones					
	T1	T2	VST3	ST3	MT3	WT3/T4	T1	T2	VST3	ST3	MT3	WT3/T4	T1	T2	VST3	ST3	MT3	WT3/T4
SorcererCC	**100**	98	93	61	5	0	**100**	99	99	86	14	0	**100**	97	86	48	5	0
CCFinderX	**100**	93	62	15	1	0	**100**	89	70	10	4	**1**	98	94	53	1	1	0
Deckard	60	58	62	31	**12**	**1**	59	60	76	31	12	**1**	64	58	46	30	**12**	**1**
iClones	**100**	82	82	24	0	0	**100**	57	84	33	2	0	**100**	86	78	20	0	0
NiCad	**100**	**100**	**100**	**95**	1	0	**100**	**100**	**100**	**99**	**6**	0	**100**	**100**	**100**	**93**	1	0

Table 6: Tool Recall and Precision Summary

	SourcererCC	CCFinderX	Deckard	iClones	NiCad
Precision	83	72	28	91	56
Precision (10LOC)	86	79	30	93	80
Recall[1]	90	75	53	78	99
Recall(T3)[2]	68	26	38	38	96

[1] Including T1, T2, VST3, ST3. [2] Including VST3, ST3.

using a minimum clone size of 10 original lines of code in order to harmonize the minimum clone size of the tools. We used the existing validation efforts, randomly selecting 30 validated clones per tool per judge (150 clones per tool) that are 10LOC or greater. These results are shown in Table 6. This precision measurement is more fair by comparing the tools under equivalent conditions, as we did with the recall experiment, but is less directly comparable with the recall results. All of the tools see a boost in precision, although NiCad most significantly. With full normalization and a generous threshold of 30% dissimilarity, NiCad may be detecting small false clones that are 6 (pretty-printed) lines or so, but contain very few tokens (spurious similarity). NiCad can be configured with a maximum clone size, and can efficiently be executed with multiple configurations, so it may be best to run NiCad with a more strict threshold for just the very small clones (6-9LOC). A full exploration of tool setting permutations versus performance is challenging and outside the scope of this paper.

5. THREATS TO VALIDITY

As observed by Wang et al. [41], clone detection studies are affected by the configurations of the tools, and SourcererCC is no exception. However, we carefully experimented with its configurations to achieve an optimal result. As for the other tools, we conducted test experiments, and also discussed with the corresponding developers for obtaining proper configurations, where available. Their configurations also provided good results in our past studies [34, 36, 37].

There are some limitations in the precision measurement. The choice of subject system (in our case a subset of IJaDataset), tool configuration [41], and targeted use-case [40] can all have a significant impact on the precision measured. The reliability of even expert judges is also a concern [4, 6, 40]. Measuring clone detection precision is very much an open problem, and although many of the obstacles in measuring precision have been identified, there does not exist a benchmark or methodology that overcomes these challenges. It is outside the scope of this work to explore new precision methodologies or benchmarks to resolve these issues.

6. RELATED WORK

Rattan et al. [25] found at least 70 clone detectors in the literature. However, very few tools target scalability to very large repositories. Liveri et al. [24] introduced a method of distributing an existing non-scalable tool to very large inputs. They partition the input into subsets small enough to be executed on a single machine, and execute the tool for each pair of partitions. Partitioning achieved scalability in execution resource requirements, while scalability in time is achieved by distribution of the executions over a large number of machines. Svajlenko et al. [34] use a non-deterministic shuffling heuristic to reduce the number of tool execution significantly at the cost of a reduction in recall. Distribution of these executions over a small number of machines is still recommended for scalability in time. SourcererCC uses a novel scalable clone detection technique, and is capable of scaling to large repositories on a single machine.

Ishihara et al. [16] use MD5 hashing to scale method-clone detection. While they achieve fast execution time, their methodology does not detect Type-3 clones, which are the most common in large repositories [33]. Hummel et al. [15] were the first to use an index-based approach to scale clone detection to large repositories, although they detect only Type-1 and Type-2 clones. Their technique produces a very large index, so the index and the computation must be distributed using MapReduce. In contrast, our SourcererCC produces a very small index, just 1.2GB for 18GB (250MLOC) of code, and detects Type-3 clones in large repositories using a single machine.

Others have scaled clone detection in domain-specific ways, and are not directly related to ours. Koshke [21] used suffix trees to scale license violation detection between a subject system and a large inter-project repository. Keivanloo et al. [20] and Lee et al. [23] use index-based approaches to scale clone search to large inter-project repositories. Chen et al. [7] implement a technique for detecting cloned Android applications across large application markets.

7. CONCLUSION

In this paper, we introduced SourcererCC, a token-based accurate near-miss clone detection tool, that uses an optimized partial index and filtering heuristics to achieve large-scale clone detection on a standard workstation. We demonstrated SourcererCC's scalability with IJaDataset, a large inter-project repository containing 25,000 open-source Java systems, and 250MLOC. We measure its recall using two state-of-the-art clone benchmarks, the Mutation Framework and BigCloneBench. We find that SourcererCC is competitive with even the best of the state-of-the-art Type-3 clone detectors. We manually inspected a statistically significant sample of SourcererCC's output, and found it to also have strong precision. We believe that SourcererCC can be an excellent tool for the various modern use-cases that require reliable, complete, fast and scalable clone detection. SourcererCC is available on our website[6].

[6] `http://mondego.ics.uci.edu/projects/SourcererCC`

8. REFERENCES

[1] Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199, 2013.

[2] Cloc: Count lines of code. http://cloc.sourceforge.net, 2015.

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

[4] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 24–49, 1992.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, Sept 2007.

[6] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère. An empirical assessment of bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 20:1–20:10, New York, NY, USA, 2015. ACM.

[7] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[8] J. Cordy. The txl programming language. http://www.txl.ca/.

[9] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 219–220, Washington, DC, USA, 2011. IEEE Computer Society.

[10] J. Davies, D. German, M. Godfrey, and A. Hindle. Software Bertillonage: finding the provenance of an entity. In *Proceedings of MSR*, 2011.

[11] D. M. German, M. D. Penta, Y. gaï£¡l Guï£¡hï£¡neuc, and G. Antoniol. Code siblings: technical and legal implications of copying code between applications. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, 2009.

[12] N. Gode and R. Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 219–228, March 2009.

[13] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *Proceedings of Working Conference on Reverse Engineering*, pages 357–366, 2012.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[15] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection:incremental, distributed, scalable. In *Proceedings of ICSM*, 2010.

[16] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 387–391, Oct 2012.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 96–105, May 2007.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, Jul 2002.

[19] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. Shinobi: A tool for automatic code clone detection in the ide. volume 0, pages 313–314, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[20] I. Keivanloo, J. Rilling, and P. Charland. Internet-scale real-time code clone search via multi-level indexing. In *Proceedings of WCRE*, 2011.

[21] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proceedings of CSMR*, pages 309–318, 2012.

[22] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 2006.

[23] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 167–176, New York, NY, USA, 2010. ACM.

[24] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of ICSE*, 2007.

[25] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199, 2013.

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

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

[28] C. K. Roy and J. R. Cordy. A survey on software clone detection research. (TR 2007-541), 2007. 115 pp.

[29] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *J. Softw. Maint. Evol.*, 22(3):165–189, Apr. 2010.

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

[31] H. Sajnani and C. Lopes. A parallel and efficient approach to large scale code clone detection. In *Proceedings of International Workshop on Software Clones*, 2013.

[32] S.Sarawagi and A.Kirpal. Efficient set joins on similarity predicates. In *Proceedings of SIGMOD*, 2004.

[33] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 476–480, Washington, DC, USA, 2014. IEEE Computer Society.

[34] J. Svajlenko, I. Keivanloo, and C. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Software Clones (IWSC), 2013 7th International Workshop on*, pages 16–22, May 2013.

[35] J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using classical detectors: an exploratory study. *Journal of Software: Evolution and Process*, 27(6):430–464, 2015.

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

[37] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, page 10, 2015.

[38] J. Svajlenko, C. K. Roy, and J. R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pages 8–9, 2013.

[39] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 495–506, 2010.

[40] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE*, pages 285–294, 2003.

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

[42] Y. Zhang, R. Jin, and Z.-H. Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.

[43] G. K. Zipf. Selective Studies and the Principle of Relative Frequency in Language.