Large-Scale Inter-System Clone Detection Using Suffix Trees

Rainer Koschke
University of Bremen, Germany
koschke@tzi.de

***Abstract*—Detecting license violations of source code requires to compare a suspected system against a very large corpus of source code, for instance, the Debian source distribution. Thus, techniques detecting suspiciously similar code must scale in terms of resources needed. In addition to that, high precision of the detection is necessary because a human needs to inspect the results.**

The current approaches to address the resource challenge is to create an index for the corpus to which the suspected source code is compared. The index creation, however, is very costly. If the analysis is done only once, it may not be worth the effort.

This paper demonstrates how suffix trees can be used to obtain a scalable comparison. Our evaluation shows that this approach is faster than current index-based techniques. In addition to that, this paper proposes a method to improve precision through user feedback and automated data mining.

***Keywords*-clone detection, code search, license violation detection**

I. Introduction

Software clone detection aims at finding equal or similar code, so called *software clones*. Software clone detection is used in software maintenance, license violation detection, plagiarism detection, code compression, code search, and other areas. It comes in different variants.

Fragment search aims at finding clones of one particular code fragment [1]–[4]. This type of code search is used, for instance, to localize code fragments where similar defects must be corrected, to find similar reusable code or examples for working solutions for a given problem, or to avoid cloning or update anomalies within an integrated development environment while a programmer is working on a particular piece of code [5]–[7].

Intra-system clone search tries to detect all clones within one given system. The primary focus is to identify opportunities for refactorings to improve maintainability, to locate code where similar changes need to be made, or to identify redundant code that can be compressed to save memory for devices with limited resources [8].

Inter-system clone search identifies all clones between a system and a corpus of other systems. As opposed to intra-system clone search, the clones within the subject system are not of interest. Typical uses cases for this type of clone detection includes management of software variants in software product lines [9], [10], finding reusable code [11], and detection of plagiarism and license violations.

All of the above variants of clone detection are facing challenges with respect to detection quality and scalability. Detection quality requires high recall and high precision in finding the relevant code. Relevance depends on the use case. In particular, inter- and intra-system clone detection need to deal with re-occurring similar code that is similar from a lexical or syntactical point of view, but that is not interesting for the given task. Frequent examples of such irrelevant similar code are import statement lists, array initializers, setter/getter sequences, or sequences of pure declarations or simple assignments.

Another challenge is scalability. While intra-system clone detection searches only within the same system, inter-system clone search may face a much larger code base, often larger by orders of magnitude. Also fragment search may face this problem, when the code is searched in very large software repositories [3], [4].

Several researchers have recently proposed to use an index-based code search to address scalability for the search in very large code bases [3], [4], [12]–[14]. Liveri et al. motivate the need for an index as follows [13]:

> "Current clone detection tools are incapable of dealing with a corpus of this size, and either can take literally months to complete a detection run, or simply crash due to lack of resources."

The index-based techniques first create an index against which code of a subject system is compared later. The purpose of the index is to identify the code that has a chance of being similar. Code filtered out by the index is not compared. Creating the index can be expensive. The idea is to invest upfront in an index that is created only once, but whose cost is amortized in multiple subsequent searches.

Contributions. This paper describes a way to extend traditional suffix-tree based clone detection for inter-system clone search that does scale for very large programs. This approach avoids the need for an index, which can be expensive to compute. In one-time analyses or when the corpus for which the index was created changes drastically, the index may not be worth the effort. Furthermore, the paper introduces a method to improve precision using software metrics and automated data mining.

II. Related Research

This section describes related research in software clone detection. It focuses on techniques using either an index

or suffix trees since these are the two approaches we are focusing on in this paper. A broader overview of clone detection in particular and software clones in general can be found in one of our earlier papers [15]–[17].

A. Techniques based on Suffix Trees

Clone detection based on suffix trees was pioneered by Brenda Baker [18]. Her technique is linear in time and space to the program length, which makes it very attractive for very large programs. It identifies consistent parameterized clones, that is, clones that are equal in the token sequence except for a consistent substitution of parameters. As *parameters*, typically identifiers and literals are considered.

The heart of the technique is a suffix tree, a very compact representation of all suffixes of the token sequence of the program. Rather than creating a suffix tree for all original tokens of the programs, Baker transforms the token sequence by summarizing all parameter tokens in one line into one artificial token representing the token type sequence of these tokens. This artificial token is called the functor. All parameter tokens of that line are added as parameters to the functor. Based on this transformation a suffix tree is constructed and re-occurring sequences are detected as clones.

The combination of several tokens into one functor reduces the number of nodes in the suffix trees. On the other hand, it makes the approach sensitive to the layout. If a programmer breaks or unites lines, the clones may not be found. As alternative, one could not summarize tokens or summarize them independently from the layout, for instance, by delimiting tokens of the underlying program languages, such as brackets or semicolons.

Token-based clone detection is known to be very fast and robust against incomplete and syntactically incorrect code [19], which makes it very attractive for inter-system clone search in large code repositories. These repositories often contain code that is syntactically incorrect. Also many languages in these repositories come with a preprocessor, which often requires to configure the code before it can be processed, and ordinary parsers would analyze only one possible configuration out of many. Token-based detection can be easily applied to the nonpreprocessed code by considering every preprocessor directive as a token. Moreover, it is relatively simple to analyze new languages with a token-based detector because only a new lexical analysis must be implemented.

In terms of quality of the results, it has high recall at the cost of lower precision because it may yield clones that do not form syntactic units [19]. For detecting license violations, higher recall is advantageous.

Token-based detectors offer some means to further improve recall by applying normalization to the token stream. For instance, visibility modifiers such as `private`, `protected`, or `public` could be removed. This approach was initially proposed by Kamiya et al. [20].

Suffix trees are not limited to tokens. We have proposed a technique that uses suffix trees for a serialization of the nodes of the syntax tree [21], [22] so that even syntactic clones can be found very efficiently.

B. Index-Based Technique

Hummel et al. were the first researchers proposing to create an index to improve performance [12]. The code is lexically analyzed and all tokens of a statement are summarized. Since the analysis avoids parsing, statement borders are recognized by delimiter tokens used to separate statements, for instance, a semicolon. Then the approach computes an MD5 hash code for chunks of code. The chunks are determined by a sliding window of fixed length over the whole code. All identifiers in a chunk are treated uniformly, that is, the approach abstracts from the exact textual representation of an identifier to generate the hash code. The length of a chunk must be the minimal length of a clone a user would use later to find clones. If that length was wrongly chosen, the index must be re-computed from scratch. The smaller the length, the longer it takes to compute the index and the larger the index becomes.

The set of MD5 hash code forms the index. If code is to be searched, the new code is processed the same way and its hash code is looked up in the index. If a match was found, the detection algorithm tries to increase the upper and lower bound of the match until it cannot be extended any further.

In their evaluation, Hummel et al. compare the index-based technique to a suffix-tree based technique. In only one of three systems, the index-based technique really outperformed the suffix-tree based technique. That happened for the largest one they considered in their study, namely, the Linux 2.6.33.2 sources with about 11 MLOC, where the technique using a suffix tree needed 166 min 13 sec and the index-based technique 47 min 29 sec.

It is interesting to note that our suffix-tree based intra-system clone detector tool, *clones*, requires only about 2 min 30 sec for the same Linux sources on comparable hardware and similar parameter settings.

Creating the index can be done in parallel on different machines to further improve performance. Hummel et al. evaluate the scale-up of the index-based technique in a distributed environment on a very large code base consisting of about 120 million C/C++ files indexed by Google Code Search, which amounts to 2.9 GLOC code. Creating the index required less than 7 hours using 1,000 computers. Then they compared a set of open-source systems (in total, 73.2 MLOC) against this index. Using 100 machines, index creation and coverage computation for these systems took about 36 minutes. For 10 machines, the processing time took slightly less than 3 hours.

Keivanloo et al. used a similar approach to gain performance for very large code bases [3], [4]. As opposed to Hummel et al., their approach also detects type-3 clones, that is, clones with added, deleted, or otherwise modified code. Their use case is fragment search in the Internet. They created an index for about 1,5 million unique Java classes (ca. 300 MLOC) from 18,000 different open-source projects. It took them about 8 hours to create the index[1]. Once this index was created, given fragments could be found in few milliseconds. They also evaluated their approach for inter-system clone detection where the whole corpus was considered to be one system. It took about 21 min to find clones between all files in their corpus.

Another index-based technique was proposed by Livieri et al. [13]. For the creation of an index for 100,000 files it took 11 h 9 min to create an index for a sliding window of 16 tokens length. The size of the resulting index was 3.7 GB.

C. Other Approaches for Large-Scale Clone Detection

Distribution and concurrency was used by other authors before to gain performance. Livieri et al. proposed to partition the clone search for very large systems into smaller pieces to be distributed among a cluster of computers [23]. If the system consists of n units (for instance, files), one can compare each unit to every other unit. Since the clone relation is symmetric and reflexive, it is sufficient to make $n \times (n-1)/2$ comparisons between units. The comparisons are then assigned to a cluster of machines, each making one comparison at a time. Each machine retrieves new comparison tasks from the pool of comparisons until all comparison are done. Then the result is a complete clone matrix. The actual comparison can be done using any type of clone detection. In their study, they used CCFinder [20]. In their evaluation, they analyzed FreeBSD with about 400 MLOC using a cluster of 80 computers in about two days.

III. Our Approach

This section describes our approach. In inter-system clone search, we are interested only in the clones between a subject system and a set of other systems, called *corpus* further on. Suffix trees have proven to allow efficient searches for type-1 and type-2 clones (identical code and code with parameter substitution, respectively). Normally, the suffix tree is generated for all tokens of a system and from this suffix tree, the clones are retrieved. This may not be possible because there are too many tokens in a huge corpus. For instance, all C files in a recent Ubuntu source distribution sum up to 1,176 billion tokens. Suffix arrays were proposed as an alternative, more compact representation for suffixes, but even for these there are ultra-large corpora whose suffix array may not fit into main memory.

[1]Personal communication with Iman Keivanloo, Oct. 25th, 2011

The basic idea to do inter-system clone detection is to generate the suffix tree for either the subject system or the corpus, whatever is smaller. We can do that because we are interested only in the clones between these two. Computing a suffix tree for a system with 10 MLOC code is a matter of one or two minutes. Further on, we assume the suffix tree is created for the subject system. Once the suffix tree is created, we take every file of the corpus and compare it against the suffix tree. This steps retrieves all subsequences of the file also contained in at least one file represented in the suffix tree, that is, the subject system. The suffix tree allows us to make this comparison in time linear to the length of the file. This will be described in more detail in Section III-A.

Many of the matching token sequences happen to be equal because they represent frequent lexical structures, which are, however, not relevant findings. Section III-C describes how we approach this problem.

A. Suffix Trees

A suffix tree is a trie representation of all suffixes of a string, $S = s_1 s_2 \dots s_n\$$, over an alphabet Σ of characters including a unique end token $\$$ and $\forall 1 \leq j \leq n : s_j \in \Sigma \wedge s_j \neq \$$. A trie is an ordered tree storing strings. In this trie, every suffix $S_i = s_i s_{i+1} \dots s_n\$$, is presented through a path from the root to a leaf. The edges are labeled with non-empty substrings. Paths with common prefixes share edges. Every outgoing edge from any node is different in its first symbol from every other outgoing edge from that node. For instance, the suffix tree for the token sequence *S = aabxyaab$* is shown in Figure 1. We label the leaves with the corresponding suffix index i when the path from the root to that leaf represents suffix S_i.

For efficient construction of a suffix tree [24], [25], the algorithms maintain a so called suffix link. A *suffix link* is an edge in the suffix tree from an internal node with path $a\alpha$ to another internal node with path α. It identifies where to add the next node during construction, but is also helpful for our purposes as we will describe shortly. The dashed edges are the suffix links in Figure 1.

As noted above, we take the token sequence T of a file and match it with the suffix tree. A naïve matching would match every suffix T_i of T always starting at the root. The complexity of this naïve matching would be $\mathcal{O}(m^2)$ with $m = |T|$, hence, this algorithm would not scale.

A linear-time algorithm is available for a somewhat more general problem, namely, computing the matching statistics [26]. The matching statistics $ms(i)$ is the length of the longest substring of T starting at position i that matches a substring somewhere in S (T is the file of the corpus and S is the whole token sequence of the subject system represented in the suffix tree). For instance, if *S = aabxyaab$* and $T = zaabtawabxpq$, $ms(1) = 0$, $ms(2) = 3$, $ms(3) = 2$, $ms(4) = 2$, etc. Clearly, there is an occurrence of S starting at position i of T if and only if $ms(i) = |S|$. That is, the

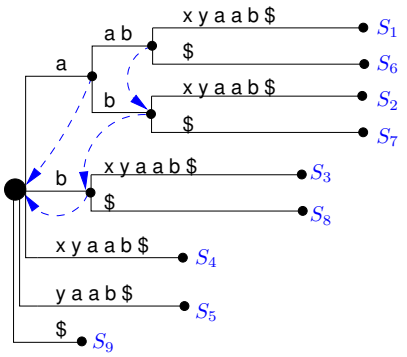

Figure 1: Suffix tree for token sequence S = *aabxyaab$*; dashed edge are suffix links

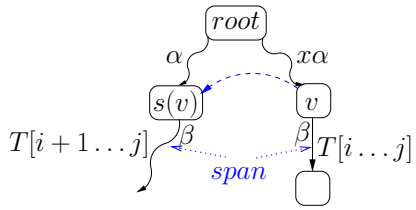

Figure 2: Suffix tree excerpt; curly edges indicate a sequence of edges; the dashed edge is a suffix link; the dotted line indicates the span

problem of finding the matching statistics is a generalization of the exact matching problem [27].

The algorithm of computing the matching statistics is sketched in the pseudocode 1. We compute the matching statistics *ms* for T from left to right, that is, for *ms(i)* for increasing $i \in 1\ldots m$ where $m = |T|$. To compute *ms(1)*, we traverse the suffix tree following the unique matching path of $T[1..m]$ as long as there are matching symbols. This step is denoted by function *path_matching* in Algorithm 1. The first argument of *path_matching* is the node at which to start and the second one is the position in T where to continue the matching. Function *path_matching* returns a point in the suffix tree for the matching path. A point in a suffix tree is an edge starting at a node v and an index of the substring denoted by the edge representing the last matching symbol on that edge. This index is called the *span*. For instance, if we were to match $T = aabxyaz$, we would reach the outgoing edge at the node v with path aab leading to the leaf S_1 in Figure 1. This edge with span 3 would be a point identifying path $aabxya$.

The length of the path *path_matching(root, 1)* is *ms(1)*. In general, let us assume we have computed *ms(i)* and want to compute *ms(i+1)* next. The previous iteration has located a point p in the suffix tree whose path matches a prefix $T[i..j]$ of $T[i..m]$ and no further match is possible. Figure 2 sketches this situation. The node reached is denoted by v. Let the part of the outgoing edge at v that was (partially) matched be the subsequence β. The span is therefore $|\beta|$. Thus, $T[i..j] = x\alpha\beta$ is the maximal matching path, where x is a single symbol and α a possibly empty string. Because a path $x\alpha\beta$ exists, there must also be a path $\alpha\beta$ in the suffix tree because it is a suffix, too. As noted above, if a node has path $x\alpha$, the node with path α can be reached by the suffix link in the suffix tree. That is, we do not need to traverse the path α from the root to identify the first matching part for the matching of next iteration's subsequence $T[i+1..j']$. Instead we take the shortcut by way of the suffix link.

Furthermore, we also know that there is a continuation β of the path to the suffix link. This part offers a shortcut, too. Again, since β must exist after $s(v)$, we can skip *span* $\leftarrow |\beta|$ symbols to continue our matching. This shortcut is a little more complicated because β may be represented by a sequence of consecutive edges reachable from the suffix link $s(v)$, while it was covered by a single edge at v.

The shortcut can take advantage of the trie property of the suffix tree. Every outgoing edge of an inner node starts with a unique symbol, so we can identify the edge to be followed easily. Let l be the number of symbols associated with this edge. If $l \leq$ *span*, we can directly move on to its target node, skipping l symbols at once. We continue with *span* $\leftarrow$ *span* $- l$. If $l >$ *span*, β was complete read, and we continue our matching at the reached edge with the current *span*.

The algorithm by Chang and Lawler [26], depicted in pseudocode 1, implements these ideas. Lines 4–12 describe what we have just said. Line 5 determines the suffix link of the node associated with the current point in the suffix tree. If the suffix link does not exist (if we are at a leaf), we backup to the parent node of the current position. Function *next* implements the skipping of symbols at the current position (subsequence β above). It yields the next point reachable from the node given as first argument, skipping the number of symbols given as second argument, and then continuing the matching of the string T the index given as third argument. Function *plen(n)* returns the path length from the root to node n, i.e., the number of symbols. It is computed and stored during suffix tree construction.

The root is a special case because it does not have a suffix link. If we are the root and the span is 0, we just continue the search from there. In this case, we cannot reuse anything from the previous search. Likewise, if its span is 1. In that case we are at the first symbol on an outgoing edge of the root. We now need to continue the search of the next symbol after that symbol, which again requires us to start at the root with *path_matching*. If, however, the span is greater than 0, we know that the first *span* number of symbols matched. There is no need to match these symbols again. So we can call *next* appropriately.

After matching has finished, we report the match (line 20) and continue with the next iteration. The reporting of

Algorithm 1 Algorithm to compute matching statistics; let θ be the threshold for the acceptable length of a clone

```
1: p ← path_matching(root, 1)
2: report_matches(T, p, 1, θ)
3: for i = 2 → m − θ + 1 do
4:    if p.node ≠ root then
5:       l ← s(p.node)
6:       if l = undefined then
7:          p ← parent(p.node)
8:          l ← s(p)
9:          n ← next(l, plen(p.node) − plen(p), i + plen(s))
10:      else
11:         n ← next(l, p.span, i + plen(l))
12:      end if
13:   else
14:      if p.span > 0 then
15:         n ← next(root, p.span − 1, i)
16:      else
17:         n ← path_matching(root, i)
18:      end if
19:   end if
20:   report_matches(T, n, i, θ)
21:   p ← n
22: end for
```

the match (function *report_matches*) receives the current position in the suffix tree and threshold θ. To obtain all matches, it just traverses the suffix tree starting at the current position to the leaves. Every leaf reached identifies a matching suffix.

A proof of correctness of the algorithm and its linear time complexity (with respect to the length of the sequence to be matched against the suffix tree) can be found in the literature (e.g., [27]).

B. Concurrency and Distribution

The search described in the previous section belongs to the class of *embarrassingly parallel problems* and can be easily parallelized and distributed. In parallel computing, an embarrassingly parallel problem is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is the case here because there exists no dependency between the comparison of files from the corpus against the suffix tree for the pattern system. Each such comparison can be assigned to an independent task.

A queue-based worker pool design pattern can be used to engage multiple independent tasks. The queue contains the work, in our case the files of the corpus to be compared. We create N different parallel tasks. Each task retrieves a work item from the queue, removes it from there, and processes it. That is, a task obtains a file, removes it from the work list, runs the lexical analysis and the comparison against the suffix tree as described above. When done, it retrieves the next file until the queue is empty. Load balancing comes for free using this approach. If N is larger than the number of physical processors, one can make efficient use of the I/O component. While one task is waiting for I/O, other tasks can process the data read.

I/O is typically the bottleneck, because – as we will see in Section IV – the actual computation is very cheap, but all the files need to be read from disk. Distribution can be used to circumvent this problem and is also easy to obtain. If one has M computers, one simply partitions the corpus into M equally large parts. The suffix tree can be computed only once and reused on all machines, but since it is so cheap to compute one could as well compute it from scratch on each machine. Then each machine processes only a portion of the corpus.

Another way to reduce the memory and CPU consumption is to summarize several tokens into one artificial token as done by Baker [18] (all tokens of one line are summarized by a functor) and Juergens et al. [28] (all tokens of a statement are summarized into one hash value). The disadvantage of Baker's summarization is that is sensitive to layout changes. The disadvantage of the summarization of Juergens et al. is that a change of a single token yields a different hash value for the statement. Furthermore, recognizing statements reliably actually requires parsing. The heuristics used by Juergens et al. on the token level may sometimes fail or may get relatively complex for syntactically richer languages such as Ada. Further research should investigate the effect of using these different approaches on runtime resource consumption and detection quality.

C. Improving Precision

We use a process leveraging code metrics and data mining to improve the precision of the findings described in this section.

The output of the above algorithm yields type-1 and type-2 clones. They are indeed alike subsequences, but many of them happen to be alike because they present frequently occurring lexical structures of a programming language. Examples for such structures are array initializers, import lists, setter/getter lists, or lists of variable declarations or simple assignment statements. One way to get rid of these is to use a filter. Our intra-system clone detector, *clones*, as well as others, for instance, the ConQAT clone detector [28], offer such capabilities. Filters can be prepared by authors of clone detectors. Preparing predefined filters by tool builders, however, requires to foresee the patterns of irrelevant clones, which is not possible in general, because they may depend upon the characteristics of the subject system and the task at hand. Creating such patterns after detection may be difficult to do for an end user and requires extra effort.

Other researchers have proposed to use code metrics to assess the quality of the found clone candidates [29]. For

every clone candidate a set of metrics is gathered characterizing the relevance of the code fragment. Thresholds of these metrics are used to filter out irrelevant clones. Typical examples of such metrics are size metrics or frequency of occurrence. To filter out uninteresting highly regular structures, such as array initializers, Higo et al., for instance, proposed specific metrics characterizing the self-repetitiveness [29]. Using this approach in practice has the problem to determine suitable thresholds. These thresholds may be calibrated experimentally by researchers. Yet, they may again depend upon system characteristics and the task at hand and hence transferability of these thresholds from one system to another is unclear.

Automated data mining techniques can be used to calibrate the thresholds from a sample. We have successfully used data mining techniques previously to determine the characteristics of interesting clones [30], [31]. Falke provides an comprehensive empirical evaluation of how data mining and software metrics can be used to exclude spurious clones [32].

We experimented with several metrics addressing the use case to identify license violations. Some of them are defined for a fragment, some of them only for a pair of cloned fragments. They represents aspects of code complexity, similarity, and self-repetitiveness.

We use *McCabe complexity MC*, that is, the number of conditions + 1. The intuition here is that code without conditions is very simple. Such code is likely not relevant for license violations.

We use two variants of similarity among two cloned fragments. *Parameter similarity PS* is the fraction of common parameters (variables and literals) in two fragments whose textual image is alike. Let $P(f)$ be the parameters of fragment f, then this metric is defined as $|P(f_1) \cap P(f_2)|/|P(f_1) \cup P(f_2)|$ for the cloned fragments f_1 and f_2.

Token similarity TS is the fraction of common tokens in two fragments as follows:

$$1 - (|(P(f_2) - P(f_1)) \cup (P(f_1) - (P(f_2))|)/|f_1|$$

(N.B.: $|f_1| = |f_2|$). This metric is similar to parameter similarity, yet sets the parameter similarity in proportion to the overall size. That is, if the clones are long and contain few parameters, the token similarity is still high even if the parameters differ more.

The metrics for self-repetitiveness address highly regular structures that are often not relevant, for instance, expressions in array initializers of the form $\{0,0,0,0,0,0,0,0,0,0,0,0,0\}$. Repetitive structures are known as repeats in stringology (the science of algorithms on strings). A *repeat* of string S is a substring of S that occurs at least twice in S. That is, our term *clone* is nothing else but a *repeat*. Essentially, we can apply clone detection to a cloned fragment to check how repetitive the cloned fragment is. We are using an algorithm by Puglisi et al. based on suffix arrays to identify all repeats with at least two tokens in a cloned fragment [33]. Based on the repeats, we use the *fractions of tokens of a fragment that are not covered by any repeat NR* as a metric for repetitiveness. For instance, if we have the token sequence $\{0,0,0,0,0,0,0,0,0,0,0,0,0\}$, then $NR = 2/27$.

Our metric *NR* is similar to the metric *RNR* by Higo et al. [29]. *RNR* can be thought of as a metric derived from *NR* and extended to clone classes.

IV. Evaluation

This section evaluates our new technique of inter-system clone search and our method to improve precision. Full data of this experiment are available online[2]. The use case in which this is to be evaluated is detection of copyright violation. We answer the following research questions:

RQ1 What are the runtime resources needed for large-scale corpora?

RQ2 What is the accuracy of our method of filtering irrelevant clones based on software metrics and data mining?

We note that our evaluation is not primarily focusing on license violations as such. We are rather trying to simulate such an exercise using large corpora to investigate the principal feasibility of our approach. Nevertheless, we do compare one commercial application against an open-source corpus to see if we can find any violation.

We refrain from comparing our approach to the index-based approaches proposed by other scientists. These techniques address slightly different use cases. Keivanloo et al. [3], [4] focus on code search of single small fragments. Hummel et al. [12] also aim at inter-system clone detection but one in which many subject systems are continuously compared to a corpus. This would support a service provider who offers license-violation detection as a service through the Internet, for instance. As we have discussed in Section II, the creation of the index can take hours even if many computers are used at once. This time may amortize if the index is reused for multiple searches. Our use case is rather that a company is analyzing its code in-house periodically but less frequently, for instance, every month or overnight.

A. Runtime Resources

This sections describes the performance evaluation.

1) Corpora and Subject Systems: We applied our new technique to the following code corpora (Table I lists size measures for these):

- **Gnome:** 132 Gnome C projects from the Ubuntu source distribution 10.04 [downloaded on 2010/17/12]
- **Ubuntu:** complete Ubuntu C source distribution 10.04 [downloaded on 2010/17/12]; this corpus includes the Gnome corpus

[2]http://www.informatik.uni-bremen.de/st/konferenzen.php?language=en

- **IJaDataset**: [4] is a set of open-source Java files originally collected by Lopes, Bajracharya, Ossher, and Baldi (known as *UCI Source Code Data Sets*, http://www.ics.uci.edu/~lopes/datasets) and then cleaned up by Keivanloo (duplicate files, files with syntax errors, and interfaces were removed) http://aseg.cs.concordia.ca/seclone [downloaded on October, 25th, 2011]

The Gnome C projects were chosen because Krinke analyzed the inter-system clones of them, too [34]. So we knew that there would be clones. Furthermore, the size of this corpus is still small enough that we can validate the detected clones.

The Ubuntu source distribution was chosen because it is a very large set of open-source programs (please note that we used only the C files in this corpus, that is, all files whose name ends with `.c`). In an exercise of detection license violations, one would certainly compare a subject system written in C to this corpus.

IJaDataset was used to investigate another programming language other than C.

As subject systems, compared to the corpora, we used the following two programs (cf. Table I for their size measures):

- **Gtksourceview**, Version 2.10.4, is a source-code view widget, written in C and part of the Gnome projects; this program is compared to the two C corpora, namely Gnome and Ubuntu
- **Commercial**, a closed-source commercial application (the name cannot be disclosed because of confidentiality) written in Java; this program is compared to the Java corpus; it was used simply because of availability; there was no suspicion that it would contain license violations.

2) Baseline: As a baseline for the comparison, we can compare our approach to the state of the practice in commercial detectors of license violations. These commercial tools often use a simple file-hashing approach. Unfortunately, these vendors rarely describe precisely how they detect violations. Often one needs to read between the lines to guess.

The file-hash comparison – our baseline – is used to find duplicated files by comparing complete files using a hash value over all tokens types of each file. It will fail when only parts of a file were copied or when the file is slightly modified. It will, however, be able to find type-2 file clones since we abstract from the parameters' textual images in computing the hash value. We use an MD5 hash function, where collisions of hash values are very unlikely.

We note that the baseline as well our approach are implemented in the same programming language. They share identical code libraries to search for files in the file system and to do the lexical analysis. They differ in only one aspect. While the baseline computes the MD5 hash and stores them in a hash table, our approach constructs the suffix tree and compares a file's tokens to it.

Subject System	Gtksourceview	Gtksourceview	Commercial
LOC	29,299	29,299	4,701,880
Tokens	139,617	139,617	32,484,058
Corpus	Gnome	Ubuntu	IJaDataset
LOC	2,346,917	182,652,865	179,855,665
Tokens	13,043,820	1,175,627,051	1,136,493,237
Our Approach			
Realtime	11 sec	10 min 46 sec	15 min 01 sec
Lexical Subject %	0.85	0.01	6.06
Suffix tree %	1.29	0.02	4.34
Matching %	97.86	99.97	89.60
max. RAM	43 MB	2.0 GB	4.0 GB
Baseline			
Realtime	7 sec	11 min 41 sec	8 min 20 sec
max. RAM	20 MB	476 MB	626 MB

Table I: Size measures (all LOC numbers exclude comments and empty lines), runtime resources

3) Platform: The results were computed on a Lenovo X220t Laptop with an Intel 64-bit 2.5 GHz CPU i5-2520M (3 MB cache) with two physical cores offering multithreading for four simultaneous threads running Ubuntu 11.04. The laptop had a SSD hard disk. 8 software threads were used. Using more than 8 threads did not further improve runtime.

4) Results: The results were gathered using 300 tokens as the minimum length for a clone. This length is justified by a court case in which the judge decided that 54 lines (out of a program with about 160 KLOC) are sufficient to be considered a license violation [35]. Table I describes size measures for the corpora in terms of lines of code and number of tokens. As the column for the Gnome projects shows, 54 lines correspond roughly to 300 tokens.

Table I lists the necessary runtime resources used to compute the results for both the new approach and the baseline technique described in Section IV-A2. CPU time is the real time under low work load (the computer ran the OS services and only the measured tool). The memory measure is the peak consumption of RAM of resident memory (the computer did not need swap space). We first measured our approach. Then the computer was rebooted to avoid cache effects.

As Table I shows both techniques are very fast, given the amount of code being processed. In case of Ubuntu, our approach is even slightly faster than the baseline. We speculate that – although both algorithms are linear – their constant factors differ. In case of the commercial subject system compared to the Java corpus, the baseline approach is much faster than our approach even though the two corpora are of similar size. The creation of the suffix tree takes longer – also in relative terms – than for *Gtksourceview*, but that alone does not explain the difference. We re-ran the analysis for a subsystem of the commercial system with almost the same number of tokens (143,650) as *Gtksourceview*. The analysis for this setting took 9 min and 9 sec. – comparable

to the time of the baseline. We speculate that the structure of the suffix tree causes this difference. Because a larger subject system implies more diversity in the input, the suffix tree will also have more nodes on the paths. More nodes on a path means that shorter code segments can be skipped during the suffix tree traversal. While the search still remains linear (by proof), the factor of the linear relation increases. Overall we note that the most costly part of the whole analysis is the lexical analysis, which requires heavy I/O.

Our approach requires more space than the baseline. The extra memory of our approach is needed for the suffix tree. This can be seen in particular for the Java analysis where the subject system is relatively large. Space consumption could be further reduced by using suffix arrays.

Overall the performance evaluation shows that our approach is equally fast as the baseline for smaller subject systems and still fast enough for larger subject systems. Our approach offers the advantage of tolerating simple changes in a file and finding copied code fragments rather than only completely identical files. Thus, it solves a more complicated problem with reasonable extra performance costs.

As another comparison, we applied our classic intra-system clone detector, *clones*, to the Ubuntu corpus. That is, it is applied to the whole the union of the subject system and corpus and then all clones not between the subject system and the corpus would need to be filtered out. *Clones* uses a suffix tree as well as a suffix array as an internal data structure. We ran this analysis on a computer with 64 GB RAM. *Clones* ran out of memory using the suffix tree, but produced a result after 23.5 hours using a suffix array. This shows that it makes sense to use a specialized approach for inter-system clone search.

The focus of the comparison of our approach with the baseline is the run-time resources needed, not the findings. Furthermore, as we noted above, our approach (as well as the baseline) will find spurious clones that are similar by accident but have no other relation. These candidates are further assessed by our approach using metrics to filter out irrelevant clones. The next section will delve into this issue in more detail. Yet, we can note here at least that the baseline approach finds only completely identical files. Consequently, its clones reported are a subset of those reported by our approach. Hence, its precision cannot be lower than the precision of our approach. Since the relevance of a clone candidate generally increases with its size and the baseline approach yields only complete units (in Java: a complete class), the precision of the baseline approach is actually expected to be higher. This potentially increased precision comes at the price of recall. If a single token is added, removed, or modified (other than renamed), the hash code will be different and the baseline approach will not find this clone. Its recall is therefore expected to be lower than our approach. We confirmed that for the Gnome study, in which our approach found pieces of copied code as well as cloned and modified files not found by the baseline.

As a side note, we point out that we did not find any copyright violation in the commercial system. However, our approach did find one class whose origin is open source. The license did allow its use. The baseline approach was unable to find it because it was slightly modified.

B. Study of Improving Precision

Our use case is detecting license violations. Any automated technique can yield only candidates for inspection. A human expert must assess the similarity, the provenance, and the legal implications. To be helpful to this human expert, the result must be precise as possible.

Because the literature on detecting license violations has described cases in which copies of fragments instead of complete files were considered violations by a judge, our aim is to provide a finer grained analysis than the state of the practice in commercial detectors. If the analysis identifies shorter clones and not only complete files, recall will increase, but precision will likely decrease. As described above, we use a filter based on code characteristics to compensate this problem.

In this section, we will investigate the accuracy of this filtering. Our focus in this part of the evaluation is the increase of precision using filtering (since it is a filter, it cannot increase recall), no longer the comparison to the baseline approach. As noted above, our approach will find all clones found by the baseline – and more, some of which possibly irrelevant.

Our filtering uses rules automatically derived from a validated sample using data mining techniques. To evaluate our approach, we looked at the results of the Gnome projects in detail. The size of this corpus is large enough for an assessment and small enough so that we can validate all clone candidates.

As above, we used 300 tokens as the minimal threshold. Our approach yielded 9,809 clone pairs and 4,668 clone classes. A clone pair is a pair of code fragments suggested as clone by the analysis. In our case, one fragment is from the subject system and the other from the corpus. Since the type-1 and type-2 clone relations are transitive, multiple clone pairs may be grouped into a clone class of all fragments that are in a mutual clone relation. Because all fragments of a clone class are alike, we inspected one clone pair in each and decided whether it is similar enough to be a relevant clone. A relevant clone in our use case would be one that is so similar that one may suspect that it has been copied from the corpus to the subject system (or vice versa). Out of the 4,668 clone classes, 3,528 (76 %) were classified as true positives and 1,140 (24 %) false positives.

We used the *rpart* component of the statistical package Rto learn a decision tree from the sample. The decision tree for the whole set of validated clones is shown in Figure 3 classifying all clones in the two categories *true* or *false* clone

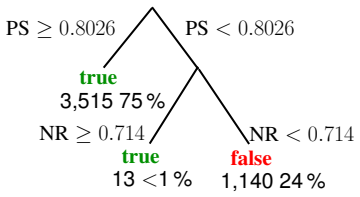

Figure 3: Learned decision tree

using code metrics. It shows that parameter similarity *PR* is the most important decision factor. All clone pairs with parameter similarity greater than 0.8 are relevant clones. At the second level, all clone pairs below that threshold are further distinguished by the fraction of tokens of a fragment that are not covered by any repeat, that is, metric *NR*. These two metrics were sufficient to classify the clone pairs. The leaves of the decision tree show the support of the rules in terms of the absolute and relative numbers of instances falling in the respective category. As Figure 3 shows, the decision tree could be further simplified by waiving the second-level rule, introducing an error of just 1 %.

The classification error of the decision tree can be assessed by the number of cases in which the rules wrongly predict the classification for a given instance (true negatives and false positives) divided by the total number of instances. The classification error for the decision tree in Figure 3 and the complete sample was below 0.001.

This decision tree was learned from the complete set of validated clone pairs. It is useful to characterize the clone pairs. In practice, one would not look at all clone pairs, but only at a subset. For this subset, a decision tree is learned, which is then applied to predict the remaining clone pairs not yet validated. To simulate this approach, we use k-fold cross-validation, which is a standard procedure in evaluating prediction models. Here, the sample is randomly partitioned into k subsamples. The decision tree is learned from $k-1$ subsamples as training data. Then the decision tree is used to predict the remaining subsample that was not used to learn the decision tree (the *test sample*). This procedure is repeated k times, where each subsample is used exactly once as test sample. Mende has shown that $k = 10$ is a useful value for evaluating defect prediction models [36], so we use that, too. The overall classification error is the mean over all classifications for all folds. This value was 0.0025 in our study, which shows a very high accuracy.

V. Conclusion

We proposed and evaluated a new approach to inter-system clone search using suffix trees, which avoids the creation of a costly index. To improve precision, the approach uses filtering based on metrics and automated machine learning. Our evaluation shows that precision can in fact be improved substantially using relatively simple metric-based filters. Furthermore, our approach is shown to be fast enough for our envisioned use case. As opposed to the state-of-the-practice clone detectors used in industry, it finds also code fragments not only complete files with affordable cost overhead.

We must note that our results depend upon the systems we analyzed and may not necessarily generalize to other systems. Also, it was our own subjective judgment whether a clone would be relevant. We plan to extend our study to additional systems and to involve other independent human oracles. Furthermore, our approach finds only type-1 and type-2 clones. We believe that this is sufficient for our use case because we expect a human expert to look at the final results. Through suitable code difference visualization, this expert should be able to combine type-1 and type-2 clones into type-3 clones. Yet, we plan to validate this assumption and possibly extend our approach to type-3 clones if needed.

Acknowledgment

I would like to thank Iman Keivanloo and Juergen Rilling for providing the Java corpus and Iman Keivanloo and Yoshiki Higo for discussing details of their techniques with me. This work has been partly funded by the DFG research grant No. KO 2342/2-2.

References

[1] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iida, "Shinobi: A real-time code clone detection tool for software maintenance," in *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2009, pp. 313–314.

[2] S. Bazrafshan, R. Koschke, and N. Göde, "Approximate code search in program histories," in *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2011, pp. 109–118.

[3] S. A. H. A. to Internet-Scale Real-Time Code Clone Search, "Iman keivanloo and juergen rilling and philippe charland," in *International Conference on Program Comprehension*. IEEE Computer Society Press, 2011, pp. 223–224.

[4] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2011, pp. 23–27.

[5] L. Barbour, H. Yuan, and Y. Zou, "A technique for just-in-time clone detection in large scale systems," in *International Conference on Program Comprehension*. IEEE Computer Society Press, 2010, pp. 76–79.

[6] M. R. Ekwa Duala-Ekoko, "Clonetracker: Tool support for code clone management," in *International Conference on Software Engineering*. ACM Press, 2008, pp. 843–846.

[7] M. de Wit, A. Zaidman, and A. van Deursen, "Managing code clones using dynamic change tracking and resolution," in *International Conference on Software Maintenance*. IEEE Computer Society Press, 2009, pp. 169–178.

[8] B. Berger and R. Koschke, "Reduzierung der Programmgröße durch Klonerkennung," in *Workshop Automotive Software Engineering, Tagungsband zur Jahrestagung der Gesellschaft für Informatik*. GI Lecture Notes for Informatics, 2008.

[9] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *Journal on Software Maintenance and Evolution*, vol. 21, no. 2, pp. 143 – 169, March–April 2009.

[10] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2008, pp. 163–172.

[11] Y. Higo, K. Tanaka, and S. Kusumoto, "Toward identifying inter-project clone sets for building useful libraries," in *International Workshop on Software Clones*. ACM Press, 2010.

[12] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "A needle in the stack: efficient clone detection for huge collections of source code," in *International Conference on Software Maintenance*. IEEE Computer Society Press, 2010, pp. 1–9.

[13] S. Livieri, D. M. German, and K. Inoue, "A needle in the stack: efficient clone detection for huge collections of source code," University of Osaka, Technical report, Jan. 2010.

[14] J. R. Cordy, "Exploring large-scale system similarity using incremental clone detection and live scatterplots," in *International Conference on Program Comprehension*. IEEE Computer Society Press, 2011, pp. 151–160.

[15] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Dagstuhl, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007

[16] ——, "Frontiers in software clone management," in *Frontiers in Software Maintenance, Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 2008, pp. 119–128.

[17] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Journal of Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009, Special Issue on Program Comprehension (ICPC 2008).

[18] B. S. Baker, "A program for identifying duplicated code," in *Computer Science and Statistics 24: Proceedings of the 24th Symposium on the Interface*, Mar. 1992, pp. 49–57.

[19] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[20] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[21] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2006, pp. 253–262.

[22] R. Falke, R. Koschke, and P. Frenzel, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

[23] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source," in *International Conference on Software Engineering*. ACM Press, 2007, pp. 106–115.

[24] E. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 32, no. 2, pp. 262–272, 1976.

[25] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica 14*, pp. 249–260, 1995.

[26] W. I. Chang and E. L. Lawler, "Sublinear expected time approximate string matching and biological applications," *Algorithmica*, vol. 12, pp. 327–344, 1994.

[27] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 2008.

[28] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective - a workbench for clone detection research," in *International Conference on Software Engineering*. ACM Press, 2009.

[29] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and implementation for investigating code clones in a software system," *Information and Software Technology*, vol. 49, no. 9–10, pp. 985–998, 2007.

[30] R. Tiarks, R. Koschke, and R. Falke, "An assessment of type-3 clones as detected by state-of-the-art tools," in *International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, 2009, pp. 67–76.

[31] ——, "An extended assessment of type-3 clones as detected by state-of-the-art tools," *Software Quality Journal*, vol. 19, no. 2, pp. 295–331, 2011.

[32] R. Falke, "Erkennung von Falsch-Positiven Softwareklonen mittels Lernverfahren," PhD dissertation, University of Bremen, Germany, 2011.

[33] S. J. Puglisi, W. F. Smyth, and M. Yusufu, "Fast optimal algorithms for computing all the repeats in a string," in *PSC*, 2008, pp. 161–169.

[34] J.Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between gnome projects," in *Working Conference on Mining Software Repositories (MSR)*. IEEE Computer Society Press, 2010, pp. 98–101.

[35] N. J. Mertzel, "Copying 0.03 percent of software code base not "de minimis"," *Journal of Intellectual Property Law & Practice*, vol. 9, no. 3, pp. 547–548, 2008.

[36] T. Mende, "On the evaluation of defect prediction models," PhD Dissertation, University of Bremen, Germany, 2011.