JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2012; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Large-scale inter-system clone detection using suffix trees and hashing

Rainer Koschke*

University of Bremen, Germany

SUMMARY

Detecting similar code between two systems has various applications such as comparing two software variants or versions or finding potential license violations.
Techniques detecting suspiciously similar code must scale in terms of resources needed to very large code corpora and need to have high precision because a human needs to inspect the results.
This paper demonstrates how suffix trees can be used to obtain a scalable comparison. The evaluation is done for very large code corpora. Our evaluation shows that our approach is faster than index-based techniques when the analysis is run only once. If the analysis is to be conducted multiple times, creating an index pays off. We report how much code can be filtered out from the analysis using an index-based filter.
In addition to that, this paper proposes a method to improve precision through user feedback. A user validates a sample of the found clone candidates. An automated data mining technique learns a decision tree based on the user decisions and different code metrics. We investigate the relevance of several metrics and whether criteria learned from one application domain can be generalized to other domains. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: clone detection, code search, license violation detection

1. 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, 2, 3, 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, 6, 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, 9, 10, 11, 12, 13].

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.

*Correspondence to: Rainer Koschke, University of Bremen, Universitt Bremen FB03, Postfach 33 04 40, 28334 Bremen, Germany, E-mail: koschke@tzi.de

Contract/grant sponsor: Deutsche Forschungsgemeinschaft (DFG); contract/grant number: KO 2342/2-2

Typical uses cases for this type of clone detection includes management of software variants in software product lines [14, 15], finding reusable code [16], 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 one 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 [13, 3, 4, 17, 18].

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. The index is a first seed of a similar code fragment. This seed is then extended by merging with neighboring similar code fragments [13, 3, 4].

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. Our conference paper introduced a way to extend traditional suffix-tree based clone detection for inter-system clone search that scales for very large programs [19]. 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 conference paper introduced a method to improve precision using software metrics and automated data mining.

This journal paper extends the original conference paper in different ways. First, we explore the benefits of creating an index to reduce the search space. Such an index may help if multiple versions of the same system are compared against the same corpus or if license violation is offered as a service where multiple different systems are to be investigated.

Second, we extend our empirical investigation of the filtering criteria and investigate whether criteria obtained from one application domain can be generalized to other domains.

Overview. The remainder of this paper is organized as follows. Section 2 gives background on related research. Section 3 describes our approach and Section 4 evaluates it. Section 5, finally, concludes.

2. 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 [20, 21, 22].

2.1. Techniques based on Suffix Trees

Clone detection based on suffix trees was pioneered by Brenda Baker [23]. 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.

DOI: 10.1002/smr

The heart of the technique is a suffix tree, a compact trie representation for all suffixes of the token sequence of the program. A *trie* is an ordered tree data structure that is used to represent strings for searching. A suffix tree is constructed for the whole token sequence and re-occurring sequences are detected as clones. More details on suffix trees follow in Section 3.1.

Token-based clone detection is known to be very fast and robust against incomplete and syntactically incorrect code[24], 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, token-base clone detection has high recall at the cost of lower precision because it may yield clones that do not form syntactic units [24]. In Section 3.4, we describe a way to improve precision not sacrificing recall.

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. [25].

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 [26, 27] so that even syntactic clones can be found very efficiently.

2.2. *Index-Based Technique*

Hummel et al. were the first researchers proposing to create an index to improve performance [13] for inter-system clone detection. 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 later use to find clones. If that length was chosen too long, the index must be re-computed from scratch when shorter clones are to be detected. 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.

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. Furthermore they study reasons for massive hash collisions. These massive collisions are typically irrelevant spuriously similar structures or recurring syntactic structures such as setters and getters or import sequences.

Another index-based technique was proposed by Livieri et al. [17]. To further limit the worst case in their clone search, they perform clone detection only at the function/method level by splitting the whole token sequence into functions, and perform the detection on each function's subsequence.

2.3. *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 [28]. 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

DOI: 10.1002/smr

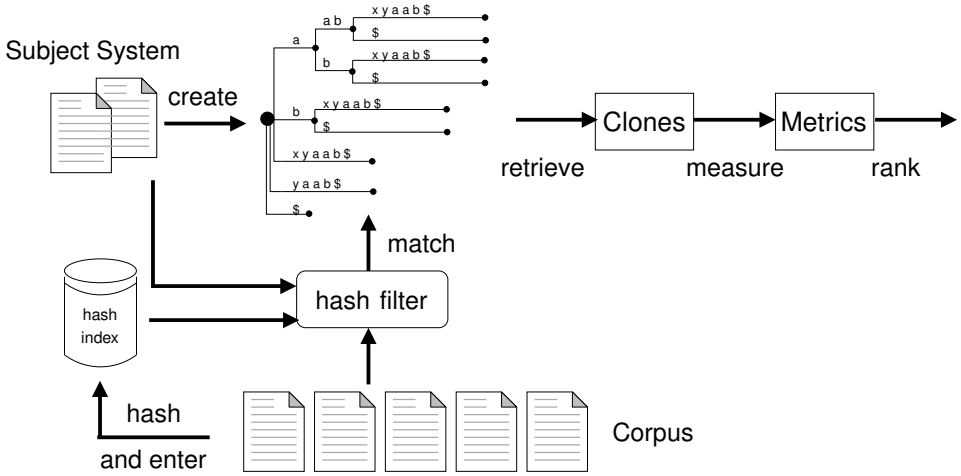

Figure 1. Major Constituents of the Detection Process

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 [25]. In their evaluation, they analyzed FreeBSD with about 400 MLOC using a cluster of 80 computers in about two days.

3. 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.

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 (further on, we assume the suffix tree is created for the subject system). Computing a suffix tree for a system with 10 MLOC code is a matter of one or two minutes. It is sufficient to compute the suffix tree for only the subject system because we are interested only in the clones between that system and the corpus. Figure 1 sketches this idea.

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 matching step will be described in more detail in Section 3.1.

Hashing as a filter can be used to reduce the number of file comparisons. If we have two files for which there is no equal hash value for any subsequence with minimal clone length, then there is no point in matching them with the suffix tree. Consequently, we create a hash index for the corpus and select a corpus file for the suffix-tree matching only if it shares a hash value with one of the subject files. This step is described in more detail in Section 3.3.

Many of the matching token sequences happen to be equal because they represent frequent lexical structures, which are, however, not relevant findings. We are measuring code characteristics of the found clones and use these metrics as a filter to improve precision. Section 3.4 delves into this processing step.

DOI: 10.1002/smr

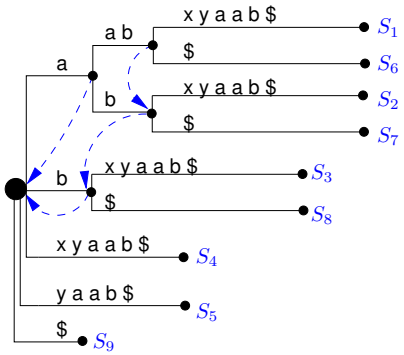

Figure 2. Suffix tree for token sequence S = *aabxyaab$*; dashed edges are suffix links

The suffix-tree based matching is the core of the comparison. Yet, there are many variants to further save space or increase speed, which we will discuss in the following sections. We briefly outline these now and then turn to suffix trees.

By summarizing multiple tokens into one supertoken, the size of the suffix tree can be reduced to save space and possibly allow faster matching. In fact, the very first suffix-tree based clone detection by Baker [9] summarized all non-parameter tokens in the same source line into one supertoken (called *functor* by Baker). Likewise the index-based technique by Keivanloo et al. [3, 4] summarizes all tokens of a line (after a syntactic pretty printing) into one supertoken. The index-based technique by Hummel et al. [13] uses certain delimiter tokens such as a semicolon to segment code and create a hash for a complete segment of tokens.

Further advances in computation time can be achieved by exploiting multi-core architectures, which are common these days. The suffix tree matching can be trivially parallelized as Section 3.2 discusses.

3.1. 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 2. 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 [29, 30], 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. Let, n be an internal node with $a\alpha$, then $s(n)$ denotes the target of the suffix link at n, referring to the node with path α. Function $s(n)$ 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 2.

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 [31]. 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 $T = zaabtawabxpq$, $ms(1) = 0$, $ms(2) = 3$, $ms(3) = 2$, $ms(4) = 1$, etc. Clearly, there is an occurrence of S starting at position i of T if and only if $ms(i) = |S|$. That is, the problem of finding the matching statistics is a generalization of the exact matching problem [32].

DOI: 10.1002/smr

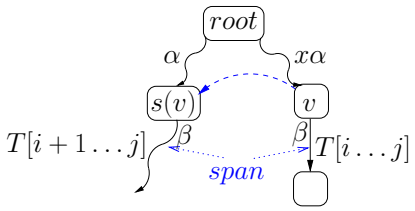

Figure 3. Suffix tree excerpt; curly edges indicate a sequence of edges; the dashed edge is a suffix link; the dotted lines indicate the span

The algorithm of computing the matching statistics is sketched as pseudocode in Algorithm 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 2. 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 3 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 use the suffix link as a shortcut.

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 completely read, and we continue our matching at the reached edge with the current *span*.

The algorithm by Chang and Lawler [31], depicted in pseudocode 1, implements these ideas. It is described also, for instance, by Gusfield [32] but only in a rather verbose manner. For this reason, we give a more detailed description here in pseudocode. 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

DOI: 10.1002/smr

Algorithm 1 Algorithm to compute matching statistics; let θ be the threshold for the acceptable length of a clone; $s(n)$ yields the node reached via the suffix link at node n

```
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 {we are at a leaf}
7:        p ← parent(p.node)
8:        l ← s(p)
9:        n ← next(l, plen(p.node) − plen(p), i + plen(l))
10:     else
11:       n ← next(l, p.span, i + plen(l))
12:     end if
13:   else {we are at the root}
14:     if p.span > 0 then
15:       n ← next(root, p.span − 1, i)
16:     else {span = 0, no symbols can be skipped}
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
```

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 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., [32]).

3.2. Concurrency and Distribution

The search described in the Section 3.1 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.

DOI: 10.1002/smr

```
int foo(int a, int i){
  while (i > 0) {
     a = a * i;
     i++;
  }
  return a;
}
```

(a) *file1.c*

```
int foo(int x, int i)
{
  while (i > 0)
  {
     x = x * i;
     i++;
  }
  return x;
}
```

(b) *file2.c*

Figure 4. Two t-identical files

I/O is typically the bottleneck, because – as we will see in Section 4 – 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 [23] (all tokens of one line are summarized by a functor) and Juergens et al. [33] (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.

3.3. Filtering by Hashing

Hashing token sequences has been successfully used by other researchers to cut the search space. In this section, we discuss ways to use such hashing in our approach. To apply hashing, we need to decide what kind of program representation, what level of granularity, and what hash function is used.

There are different program representations upon which the hash value can be computed. We opt for tokens rather than characters. The latter is subject to layout and comment differences. Also, the clone detection is based on token types and, hence, the hashing should be based on the same representation. In particular, through the token representation, we can abstract from parameters analogous to the suffix-tree based search. That is, we ignore the concrete token value of an identifier or literal and consider both as a parameter token so that we can cope with type-2 clones.

We can distinguish two levels of granularity for hashing: file level and token level. In file-level hashing, all tokens types of a file are hashed. Two files with the same hash are likely alike (depending upon the strength of the hash function). More precisely, we consider two files, f_1 and f_2, *token-identical*, or *t-identical* for short, if the hash values of their token type sequences are the same (*tokens*(f)) denotes the token-type sequences of f):

$$\textit{t-identical}(f_1, f_2) \Leftrightarrow \textit{hash}(\textit{tokens}(f_1)) = \textit{hash}(\textit{tokens}(f_2))$$

The two files shown in Figure 4 have the same token type sequences and are, hence, t-identical. The token-type sequence abstracts from the layout and parameter differences between the two files.

Hashing of complete files is helpful to clean up the corpus, which may contain identical files, for instance, because it contains multiple versions of the same software. Identical files can be identified by these hashes, recorded somewhere, and then deleted to avoid duplicate comparisons. File-level hashing is also helpful in the comparison between the pattern system and the corpus, as it will identify all t-identical files. These files need not be compared using the suffix tree.

DOI: 10.1002/smr

Token-level hashing computes hashes for token sequences within a file. Such token subsequences are also known as n-grams. An n-gram is a sequence of consecutive tokens with length n within a file. If the length of a file is N, then we would have $N - n + 1$ hash values for this file while there is only one hash value for each file. Thus, token-level hashing is much more costly than file-level hashing both in terms of time and space resources needed. The value n is chosen equal to the minimal clone length that is to be searched for. If a file a and a file b do not have two n-grams, $\textit{n-gram}_a \in a$ and $\textit{n-gram}_b \in b$, respectively, for which $h(\textit{n-gram}_a) = h(\textit{n-gram}_b)$ – where h is the hash function – then they do not share any code of at least length n, and this pair of file can be excluded from the comparison using the suffix tree.

Token-level hashes can be created for the corpus before the comparison of subject system and corpus and persistently stored as an index. If a subject system is to be analyzed, it is hashed in the same way. Then we can retrieve the corpus hashes from the index. We select and match only those corpus files that share at least one hash value with any of the subject-system files. This filter may reduce the number of corpus files to be processed. In addition, we can add only those subject-system files to the suffix tree that share a hash value with any corpus file, which can reduce the suffix-tree size and the time needed for the matching because the suffix tree can be more quickly traversed.

Another variation is the choice of a concrete hash function. For file-level hashing, we use MD5 because collisions are extremely unlikely. For token-level hashing, we select CRC hashing. CRC is not as strong as MD5, but easier to compute. Because the token-level hashing is used only to exclude files from the search for which we will never find a match, collisions are not a real problem. If two files happen to have n-grams with the same CRC, we will compare them using the suffix tree. If they are in fact different, the suffix tree will tell us and we just made an unnecessary comparison. The opposite cannot be true. If two files do not share n-grams with the same CRC hash, the suffix tree will equally not find any shared token sequence. This is true for every deterministic hash function.

One might argue that the suffix tree is not really needed anymore if one has the hash index already. The purely index-based techniques try to extend a found hash value by preceding and subsequent neighboring hash values to identify larger clones than the minimal clone-length threshold used to create the index. They can find clones without any suffix tree just on the basis of the hashes. Yet, to do that the index must store the order in which the hashes appear. To locate the clones in the code, the index must also store the position information of each hash in the source code. Our approach neither needs to store the order nor any location because only the existence of a shared hash counts. Also, if there are several n-grams with the same hash, only one of them needs to be stored in our approach. For this reason, the index requires less space in our approach. A possibly more important argument is the flexibility of our hybrid approach. The index must be created with a fixed minimal clone length. Later clones smaller than that cannot be detected. In our case, however, the clone length could in fact be chosen smaller when we use the suffix-tree matching. That is to say, we would have two levels of minimal clone lengths. The first one identifies all pairs of files that share a clone of minimal length chosen to prepare the index. Yet, if such a pair was identified, we could choose a shorter clone length to find more clones between these files using the suffix tree. This could be very helpful in type-3 clone detection where smaller type-1 or type-2 clones are merged into larger type-3 clones. It may also be helpful if the goal is to measure the similarity of files if one wants to raise the level of abstraction. We could design similarity metrics on the basis of shared code among files. Here again the suffix tree could be used to identify the smaller clones once one larger clone was identified by the hashing.

3.4. 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,

DOI: 10.1002/smr

for instance, the ConQAT clone detector [33], 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 [34]. 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 [34]. 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 [35, 36]. Falke provides an comprehensive empirical evaluation of how data mining and software metrics can be used to exclude spurious clones [37].

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 represent 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 follows for two cloned fragments f_1 and f_2:

$$PS(f_1, f_2) = |P(f_1) \cap P(f_2)|/|P(f_1) \cup P(f_2)|$$

For instance, $P(file1.c) = \{foo, a, i, 0\}$ and $P(file2.c) = \{foo, x, i, 0\}$ in Figure 4. Then $P(file1.c) \cap P(file2.c) = \{foo, i, 0\}$ and $P(file1.c) \cup P(file2.c) = \{foo, a, x, i, 0\}$, and $PS(file1.c, file2.c) = 3/5$.

Token similarity TS is the fraction of common tokens in two fragments as follows ($\triangle$ denotes the symmetric difference $A \triangle B = (A \cup B) \setminus (A \cap B)$):

$$1 - |P(f_1) \triangle P(f_2)|/|tokens(f_1)|$$

Please note that $|tokens(f_1)| = |tokens(f_2|)$ because f_1 and f_2 form a type-1 or type-2 clone.

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.

For instance, $P(file1.c) \triangle P(file2.c) = \{a, x\}$ and $|tokens(file1.c)| = 31$. Thus, $TS(file1.c, file2.c) = 1 - 2/31 \approx 0.94$.

To exclude any misunderstanding, we point out that the examples illustrating the definition of *PS* and *TS* use the two complete files in Figure 4, although the metrics are defined for every token subsequence, not just those for complete files; in the example of Figure 4, the cloned token sequence just happen to span whole files.

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 [38]. Based on the repeats,

DOI: 10.1002/smr

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. [34]. *RNR* can be thought of as a metric derived from *NR* and extended to clone classes.

4. EVALUATION

This section evaluates our new technique of inter-system clone search and our method to improve precision.

In this section, we evaluate both our new technique of inter-system clone search and our filtering approach to further improve precision. The evaluation was performed for the use case of detecting copyright violations. All data used in these experiments are available online[†]. We answer the following research questions:

RQ1 Does inter-system clone detection based one suffix trees scale to very large corpora?
RQ2 What is the effect of filtering by n-gram hashes?
RQ3 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. That is to say, in our study we do not actually search for license violations as we are comparing open-source projects to open-source projects and we do not further investigate whether the clones found represents true violations of license restrictions. There is much more to that than finding similar code. It is foremost a legal issue. The setting in the evaluation just mimicked this use case in comparing large programs to a very large corpus to investigate the principal feasibility of our matching approach.

4.1. Comparison to Other Approaches

We consciously refrain from directly comparing our approach to the index-based tools developed by Hummel et al. [13], Livieri et al. [17], and Keivanloo et al. [3, 4] because they address incomparable use cases. Keivanloo et al. focus on code search of single small fragments while Hummel et al. as well as Livieri et al. [17] primarily aim at intra-system clone detection. That is, we do not use their tools in our study, also because these are implemented with different technologies and programming languages, which makes a fair comparison difficult. On the other hand, We are aware that the existing index-based techniques by Hummel et al., Livieri et al. and Keivanloo et al. could be easily extended to inter-system clone detection, too. That is why we do explore the benefits of their core idea, namely, using a hash-based index. We do that by integrating a hash index into our approach and evaluate its effect on performance. By implementing hashing ourselves on the basis of our implementation infrastructure, we also get results that are more meaningful than comparing tools based on disjoint implementations.

Although we do not use their tools in our evaluation, we at least summarize their performance studies here briefly. 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. Apparently, the reason for this performance discrepancy must lie in implementation issues,

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

DOI: 10.1002/smr

which supports our argument that comparing tools written in different languages and using different implementation strategies can be problematic.

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. 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, which is a setting of inter-system clone detection. 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]. 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[‡]. 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. [17]. 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.

4.2. Runtime Resources

This sections describes the performance evaluation.

4.2.1. Corpora and Subject Systems We applied our new technique to the Ubuntu 10.04 source corpus (downloaded on 2010/17/12) consisting of only C files having the file extension `.c`. All files with other extensions were deleted to avoid affecting the speed of the actual directory traversals. This corpus consists of 420,113 C files.

The Ubuntu source distribution was chosen because it is a very large set of open-source programs. In an exercise of detection license violations, one would certainly compare a subject system written in C to this corpus.

The Ubuntu corpus is structured into different directories, where each directory holds the code for one project. In some cases, there are directories for different versions of the same projects. For instance, the Linux kernel versions 2.6.31 and 2.6.32 are both contained. For this reason, it makes sense to clean up the corpus by removing duplicate files using the hashing approach outlined in Section 3.3. We found 117,393 t-identical files that we removed from the scope of the analysis. That is, 28 % of the original files of the corpus could be removed. The time needed to identify these t-identical files was 16 min 33 sec. The 117,393 identical files fell into 39,339 file clone classes, that is, some files appeared more than twice. Most of these duplicated files are in directories for different versions of a system. Occasionally, there are also duplicated files between different projects. The t-identical files in one clone class were lexically sorted by their name and only the first one in the ascending order was kept.

We will use the Ubuntu corpus for our performance studies. In the study of improving precision using data mining techniques, we will use automated machine learning to derive a decision model. This decision model will be learned from only a subset of the Ubuntu corpus, which consists of 132 different Gnome projects.

4.2.2. 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 12.04. 8 software threads were used. Using more than 8 threads did not further improve runtime. The laptop had a SSD hard disk, which helps in disk-intensive applications

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

DOI: 10.1002/smr

Subject System	#files	LOC	SLOC	#tokens
gtksourceview-2.10.4	35	39,438	28,105	134,325
zabbix-1.8.1	196	98,391	67,827	358,908
dvdrtools-0.3.1	194	109,115	86,910	573,623
cluster-3.0.2	328	198,513	153,183	800,410
netw-ib-ox-ag-5.36.0	717	238,057	201,942	994,797
opensaf-4.0.B4	645	573,543	418,558	2,023,146
mesa-7.7.1	1,236	777,799	614,332	3,528,098
kompozer-0.8b1	1,441	1,233,012	975,104	4,905,588
wine-1.0.1	2,076	2,042,649	1,636,239	9,872,157
sdlmame0136	2,695	2,539,350	2,050,290	13,713,827
linux-2.6.32	12,042	8,825,447	6,819,045	37,440,842

Table I. Size measures (LOC = lines of code, SLOC = lines of code excluding comments and empty lines) after corpus clean-up

such as ours, and 8 GB of RAM were available. During all time measurements, not more than 3,984 MB Virtual Image (VIRT) was used. VIRT is the total amount of virtual memory used by the program, including all code, data and shared libraries plus pages that have been swapped out. Yet, the operating system never needed to use swap space on disk in our study.

4.2.3. Results In our previous conference paper, we used 300 tokens as the minimum length for a clone. This length was 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 [39]. The average token density for Linux, for instance, is 5.5 tokens per SLOC and 54 SLOC then roughly correspond to 300 tokens. In the light of more recent court cases (e.g., Oracle versus Google) where even shorter clones were found to be relevant, we further lowered the threshold to 100 tokens (roughly 8 SLOC) in our new study.

Table II lists the necessary runtime resources used to compute the results for the subject systems listed in Table I. CPU realtime is the real time in milliseconds under low work load (the computer ran the OS services and only the measured tool). The last column lists the number of clone pairs found. All other columns show the fraction of runtime needed for the different steps. Step *lexical analysis* consists of searching the files of the subject system in the directory tree and the lexical analysis of all found files. Step *suffix tree* is the creation of the suffix tree for the subject system. The *matching* step contains the time needed to find the corpus files in the directory tree of the corpus, run the lexical analysis on each found file, do the comparison against the suffix tree, and report the results. Step *others* subsumes every other activity such as start-up and clean-up of the tool. The ordering of the table rows is consistent with the one in Table I, namely, the ordering by size in terms of number of tokens.

Table II shows that the technique is very fast, given the amount of code being processed. As expected, the runtime is primarily dominated by the corpus size. In the range of the subject systems from gtksourceview-2.10.4 to mesa-7.7.1, which span a token size range from ca. 130 thousand tokens to 3.5 million tokens, the runtime is roughly the same. For the larger systems except for, kompozer-0.8b1, the runtime increases only somewhat. Here we expectedly find many more clones, which means that more time is needed for matching and reporting. Also, the suffix tree is larger for these programs, hence, more time is needed to construct it. We can expect that the suffix trees is more diverse because of the size of these systems show more diverse token sequence patterns. More diversity in the suffix tree means again that it has more tree nodes and the paths covered by suffix tree edges are shorter, hence, only 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 and more absolute time is needed to traverse the tree.

An interesting outlier is kompozer-0.8b1. It belongs to the larger systems, but that alone cannot explain why the analysis took so long. It has only about half of the tokens wine-1.0.1 has, but

DOI: 10.1002/smr

system	realtime	lexical subject	suffix tree	matching	others	#clones
gtksourceview-2.10.4	1,152,464	0.02 %	0.01 %	99.96 %	0.01 %	12,810
zabbix-1.8.1	1,192,956	0.05 %	0.03 %	99.90 %	0.02 %	30,709
dvdrtools-0.3.1	1,195,160	0.04 %	0.03 %	99.91 %	0.02 %	356,949
cluster-3.0.2	1,193,412	0.08 %	0.07 %	99.84 %	0.01 %	144,232
netw-ib-ox-ag-5.36.0	1,184,260	0.13 %	0.09 %	99.77 %	0.02 %	465,091
opensaf-4.0.B4	1,198,451	0.28 %	0.18 %	99.52 %	0.01 %	206,172
mesa-7.7.1	1,287,461	0.33 %	0.36 %	99.30 %	0.01 %	2,250,411
kompozer-0.8b1	5,180,733	0.12 %	0.12 %	99.75 %	0.01 %	33,017,646
wine-1.0.1	1,653,609	0.69 %	0.66 %	98.63 %	0.01 %	15,735,806
sdlmame0136	1,643,744	1.11 %	1.52 %	97.36 %	0.01 %	20,757,374
linux-2.6.32	1,508,530	1.86 %	2.15 %	95.98 %	0.01 %	7,006,761

Table II. Runtime in milliseconds

required ca. three times longer to be analyzed. The reason for this lies within the structure of this code. The code contains many re-occurring highly regular code structures, such as array initializers, case label sequences, *#define* sequences. These highly repetitive code fragments are not only often re-occurring in other systems but they also create many overlapping clone candidates. Suppose $T = aaaaa$ is a token sequence in the subject system and there is another sequence *aaa* in a corpus file. Then we have three clones of the subsequence *aaa* between the two. The first starts at the first a of T and ends at the third position. The second starts at the second position and ends at the fourth. The third ranges from the third to the last a in T. Such self-repetitive structures increase the number of cloned candidates a lot. They pose a general problem on token-based clone detection and motivates the repetitiveness metrics described in Section 3.4.

Table II also breaks down the time needed for the different steps of the analysis. Lexical analysis and suffix tree construction is negligible even though it tends to be higher for the larger systems. Step *matching* is dominating and within that step the most costly part is the lexical analysis, which requires heavy I/O.

As another comparison, we applied our classic intra-system clone detector, *clones*, to the Ubuntu corpus. That is, it was applied to the union of the subject systems and corpus. By filtering out all clones not between a subject system and the corpus, one could use a standard intra-system clone detector for inter-system clone detection.

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.

4.2.4. Effect of Filtering by Hashing To compute the index, n-grams of length 100 tokens were hashed using a 32-bit CRC. The total creation of the index – including lexical analysis, hashing, and storing on disk – took 36 min 54 sec. The size of the created index file on disk was 6,568 MB. As a point of comparison, matching the largest subject system linux-2.6.32 with the corpus without the index took about 25 min. That is, the index makes sense only if we run the analysis several times against the same corpus.

The effect of the filter can be seen in Table III. The columns are similar to Table II. The difference here is that column *lexical subject* now contains not only the lexical analysis of a subject file but also the creation of the n-gram hashes for subject files and reading and browsing the index file, required to search for all corpus files that share any of these indexes. That is why this step is more expensive than without using the index. As discussed in Section 3.3, we could construct the suffix tree only for those subject files that pass the filter to save space and time on the suffix tree. We did not do that in this study because that would have required more invasive changes in our current implementation and suffix tree construction is relatively cheap. Consequently, the results reported here are a lower bound of the savings using hashing as a filter.

DOI: 10.1002/smr

system	realtime	lexical subject	suffix tree	matching	others	ignored
gtksourceview-2.10.4	95,437	0.69 %	0.14 %	66.15 %	33.02 %	80.3 %
zabbix-1.8.1	152,539	1.18 %	0.22 %	68.07 %	30.52 %	79.2 %
dvdrtools-0.3.1	161,118	1.36 %	0.27 %	59.77 %	38.60 %	78.6 %
cluster-3.0.2	178,945	2.11 %	0.46 %	62.64 %	34.79 %	77.2 %
netw-ib-ox-ag-5.36.0	198,749	2.49 %	0.54 %	63.68 %	33.29 %	76.9 %
opensaf-4.0.B4	220,008	4.80 %	1.00 %	61.85 %	32.35 %	74.6 %
mesa-7.7.1	271,062	6.15 %	1.73 %	66.87 %	25.24 %	72.5 %
kompozer-0.8b1	2,106,924	1.18 %	0.32 %	95.61 %	2.89 %	71.4 %
wine-1.0.1	473,371	9.31 %	2.34 %	75.44 %	12.91 %	68.9 %
sdlmame0136	567,759	11.52 %	4.51 %	71.54 %	12.43 %	68.0 %
linux-2.6.32	484,839	22.29 %	6.72 %	52.37 %	18.62 %	65.6 %

Table III. Runtime (in milliseconds) using hashing as a filter; column *ignored* lists the percentage of the 420,113 corpus files that were filtered out for the comparison using the index

As can be seen in Table III in comparison to Table II, the *matching* step is now relatively cheaper. The reason is the great effect of the filter, which requires fewer corpus files to be scanned and matched. The effect of the filter is shown in the last column of Table III, which shows the number of files that did not pass the filter and were excluded from scanning and matching. We see that this filtering is quite effective. In case of kompozer-0.8b1, the filter reduces the search by half, but that system has so many clone candidates and, hence, the filter's effect is expected to be lower. It is interesting that the filter itself has similar effect as for the other systems, that is, it excludes as many corpus files as for the other systems (we note the trend that the filter gets less effective for larger systems). Consequently, the similarity between kompozer-0.8b1 and the corpus concentrates on a smaller subset of the corpus files, producing a large amount of clone candidates.

Overall, this performance study has shown that suffix-tree based inter-system clone detection is very fast. It can be further made faster by using a hashing filter when the same corpus is compared more than once. If the analysis is run only once, it is cheaper to avoid the hash index.

4.3. Study of Improving Precision

One of our envisioned use cases 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 as 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.

While our earlier study in the original conference paper [19] focused on the accuracy of this filtering, the study presented here additionally investigates whether the filter learned for one particular sample can be generalized to others.

4.3.1. Accuracy Measures Our focus in this part of the evaluation is the increase of precision using filtering based on code characteristics. Because it is a filter, it cannot increase recall. By design of the algorithms (proven correct by the original authors), we are guaranteed to find every pair of token sequences (one of which of the subject system and the other of the corpus) of a given minimal length that are t-identical. That is, with respect to this matching, our algorithm provides 100 % recall. Recall can only be lower than that by defects in our implementation (which are of course possible) and by a more general notion of recall: From a practical point of view, we can argue that we also want to find cases where a programmer slightly modified the copied code. She or he could do so by adding or removing tokens, in which case the clone would be of type 3. Every type-3 clone consists of multiple type-1 or type-2 clones. If these are still longer than the minimum

DOI: 10.1002/smr

threshold, our approach would still find them. Vice versa, if they are shorter, our approach will miss them. To some extent, we could compensate for that by lowering the minimum clone length and then merging smaller consecutive type-1 and type-2 clones into type-3 clones as others have done [40, 41, 42]. Yet, even this approach requires a minimum threshold for the initially detected type-1 and type-2 clones and may, hence, fail. In addition to that, a programmer could apply more substantial transformations to the copied code, for instance, to obfuscate her or his act of copying. Although such obfuscation might be counteracted by normalization to some extent, it is unlikely that we can de-obfuscate all kinds of modifications. This problem is well known in the world of virus detection. The question in our setting is, how likely obfuscation may occur in code re-use. The programmer copies code to save time now and also in future maintenance. Merging changes with new enhancements or improvements in the original code is more difficult when the copied code was obfuscated. In summary, our technique will find all t-identical clones of a given minimal length; it may fail only if the copy is modified. Detecting modified clones reliably is still an open problem in general.

Our focus here is rather to increase the relevance of detected t-identical token sequences by ranking clone candidates with respect to code characteristics and similarity measures. When a cut-off is used for a given ranking, some clones will be excluded. In this setting, recall can be sensibly measured as the fraction of real clones above the cut-off divided by the total number of real clones found. That is to say, we measure recall of the filter and not of the underlying clone detection technique.

Before we delve into the details of the accuracy measures to evaluate our filter, we introduce some terminology. The pairs of t-identical token sequences detected by the clone detector are called *clone candidates*. If a human decides that a clone candidate is in fact a clone, it becomes a *real clone*. Multiple t-identical token sequences can be summarized into a *clone class*, when they are all t-identical. By construction, a clone class has always at least one token sequence from a file of the subject system and one from a file of the corpus. Because all token sequences are alike with respect to the notion of t-identical, they may be validated together by a human.

We use recall and precision as measures of accuracy. The decision tree is a binary classifier that divides clone candidates into two disjoint sets: TCC consisting of the supposedly true clone candidates and FCC consisting of supposedly false clone candidates. A human oracle acts as a binary classifier, too, dividing clone candidates into real clones RC and false clones FC. Given these two separate classifications, we can define the following differences and agreements (the so called *confusion matrix*):

- true positives, $TP = RC \cap TCC$, i.e., real clones that were correctly classified as clones by the filter
- false negatives $FN = RC \cap FCC$, i.e., real clones that were incorrectly marked as false clones by the filter
- false positives $FP = FC \cap TCC$, i.e., false clones that were incorrectly labeled as clones by the filter
- true negatives $TN = FC \cap FCC$, i.e., false clones that were correctly labeled as false clones by the filter

Recall is then defined as $|TP|/(|TP| + |FN|)$ and precision as $|TP|/(|TP| + |FP|)$.

In addition to that, the classification error of a decision tree is typically 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: $(|TN| + |FP|)/(|TP| + |FN|)$.

4.3.2. Subject Systems 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 Gnome C projects were chosen because Krinke analyzed the inter-system clones of them, too [43]. So we knew that there would be clones. Furthermore, the size of this corpus is large enough for an assessment and small enough so that we can validate all clone candidates.

DOI: 10.1002/smr

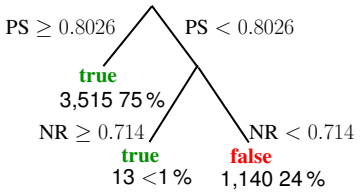

Figure 5. Learned decision tree

There are 132 Gnome C projects contained in the Ubuntu source distribution 10.04 (downloaded on 2010/17/12). It consists of 2,346,917 SLOC or 13,043,820 tokens. These projects were also part of our performance study in the previous section.

We repeat our study of the Gnome systems – originally published in our earlier conference paper [19] – here so that this journal paper is self-contained. At the time of the research for the conference paper, we followed the 54-line threshold of the court case described above, hence, used 300 tokens as the minimal threshold.

4.3.3. Learning a Decision Model Our approach yielded 9,809 clone pairs and 4,668 clone classes. 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 for human inspection. 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) and should look more closely. Whether it is in fact a copy or even a true license violation was not the driver of our decision. So even if code was automatically generated (and we found some of them occasionally), we would classify it as a clone if it was so similar that a human expert should look at it. The context of the supposedly cloned code or comments therein often gave us additional hints. Similar context and similar or even identical comments were treated as strong indicators for cloning.

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 R[§] to learn a decision tree from the sample. The decision tree for the whole set of validated clones is shown in Figure 5 classifying all clones in the two categories *true* or *false* clone using code metrics. Its classification error for the complete sample is below 0.001.

Figure 5 shows that parameter similarity *PS* 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 5 shows, the decision tree could be further simplified by waiving the second-level rule, introducing an error of just 1 %.

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 [44], so we use that, too. The overall classification error is the mean over all classification errors for all folds. This value is 0.0025 in our study, which shows a very high accuracy.

[§] http://www.r-project.org/

DOI: 10.1002/smr

4.3.4. Transferability the Decision Model While the evaluation of accuracy so far was conducted on a more homogeneous code base, namely, various Gnome projects, we now investigate whether the decision tree learned for this sample can also be applied to other systems outside this Gnome family. We randomly selected clone pairs from different projects of the Ubuntu corpus outside of the Gnome family on which the decision model was calibrated. These clone pairs were selected from the results obtained for the systems listed in Table I excluding *gtksourceview-2.10.4*. The system *gtksourceview-2.10.4* is excluded because it belongs to the Gnome projects and was used to learn the decision model as described in the previous section. We selected 210 clone pairs randomly and evenly from all considered systems.

As opposed to the calibration of the decision model described in the previous section, in this part of the evaluation we selected 100 tokens as the minimum clone length to further mitigate the external threat to validity, namely, the dependency on the minimum clone threshold. Earlier we used 300 tokens as the minimal clone length, which has a higher chance to detect clones with high precision. Shorter minimum clone length tends to decrease precision. Consequently, the decision tree is challenged not only by a new system with potentially different characteristics but also by shorter clone candidates, which are typically less precise.

The author of this paper looked at each pair in this sample and classified it as real or false clone based on the same guideline used in Section 4.3.3 to calibrate the decision model. There was only one clone pair among all these 210 randomly selected clone pairs that was considered a real clone. The parameter similarity *PS* of it was 1. The maximal *PS* observed for all other clone pairs was 0.667. For this maximum, the fraction of non-repetitive tokens *NR* was 0. Consequently, the decision tree would correctly classify the real clone as true clone candidate and would reject all false clones.

The result shows that the precision of the underlying clone detection (not the filter) is very low for token sequences of at least 100 tokens and a filtering is urgently needed. Yet, we cannot generalize from only one positive example. For this reason, we complemented the completely random sampling by a quota random sampling where we partition the data and have a fixed quota of drawings instances from each partition. Our previous sample had 190 clones pairs whose *PS* were lower than 0.1, 4 clone pairs in the range $[0.1, 0.3)$, 3 in $[0.3, 0.4)$, 2 in $[0.4, 0.5)$, 7 in $[0.5, 0.6)$, 3 in $[0.7, 0.1)$ and only one pair had *PS* = 1.0. For this reason, we decided to look more closely in the range of 0.5 and 1.0 for *PS*. We partition this range into intervals with a step of 0.05 and attempt to select 10 clone pairs (one from each subject system listed in Table I except for gtksourceview-2.10.4) from each of these 10 partitions for validation. We did not find any clone pair in the partitions $[0.55, 0.6)$, $[0.7, 0.75)$, $[0.75, 0.8)$, $[0.8, 0.85)$, $[0.85, 0.9)$, and $[0.9, 0.95)$ for opensaf-4.0.B4 and none in the partition $[0.55, 0.6)$ for zabbix-1.8.1. Consequently, our new quota sample consists of $10 \times 10 - 7 = 93$ clone pairs.

These clone pairs were again validated by the author. This validation provides us with two sets, RC and FC, introduced in Section 4.3.1.

After that, the decision model is applied to $RC \cup FC$ yielding TCC and FCC. Based on that, we can measure recall and precision. We obtained $|TP| = 37$, $|FP| = 4$, $|FN| = 6$, and $|TN| = 46$. Hence, recall is 0.86 and precision is 0.90.

As another point of comparison, we used the automated machine learning again to derive a decision model for our quota sample consisting of these 93 clone pairs. The derived rules are this time as follows:

1. a clone candidate is a real clone if *NR* ≥ 0.234465
2. if a clone candidate has *NR* < 0.234465, then
   (a) it is a real clone if *PS* ≥ 0.83639
   (b) it is a false clone if *PS* < 0.83639

Again only *NR* and *PS* were used although all metrics were given as input to the automated learner. This time, *NR* is more important and has a different value than in the Gnome study, while *PS* is very similar to the value obtained in the Gnome study. The reason why *NR* gained more importance is the fact that the Ubuntu sample contained many more repetitive structures than the Gnome sample.

DOI: 10.1002/smr

This result indicates that the relevance of a parameter can vary from system to system, which challenges transferability. Despite this difference, however, both decision trees used the same types of parameters (*NR* and *PS*), and even the original decision tree showed a good accuracy for the new subject system.

4.3.5. Qualitative Analysis and Possible Improvements To get more insight into the differences, we looked specifically at the false positives and false negatives. We want to understand why the Gnome decision model applied to the complete Ubuntu corpus did not always work and how it could be improved.

FN_1 One false negative is an if-then-else cascade whose inner blocks are alike – hence, a highly regular structure – except for one consistently replaced identifier (`bi` in one fragment and `bits` in the other fragment) that occurs in almost every line – hence a low parameter similarity according to *PS*. One of the cloned fragments is given here (differences are highlighted):

```
bi->bi_flags = 0;
if (length == 1)
{
  bytes = bytes_left;
  bi->bi_offset = sizeof (struct gfs2_rgrp);
  bi->bi_start = 0;
  bi->bi_len = bytes;
}
else if (x == 0)
{
  bytes = sdp->sd_sb.sb_bsize - sizeof (struct gfs2_rgrp);
  bi->bi_offset = sizeof (struct gfs2_rgrp);
  bi->bi_start = 0;
  bi->bi_len = bytes;
}
else if (x + 1 == length)
{
  bytes = bytes_left;
  bi->bi_offset = sizeof (struct gfs2_meta_header);
  bi->bi_start = rgd ...
```

This shows a point of improvement for our definition of parameter similarity. It could be improved by not just comparing positionally but by the bindings of re-occurring parameters similarly to the original approach by Baker [9]. That is, when one identifier a is replaced by another identifier b, we should expect b at all position where a appeared.

FN_2 A case-cascade falls into the category of false negatives for the same reason as FN_1. One identifier was systematically replaced.

FN_3 FN_3 also belongs to the category of FN_1 and FN_2. The clone is a complete function body where two parameters were renamed systematically. Interestingly, the function signature was also equivalent, but one used the old K&R C listing of parameters and the other the new style. To detect this, one would need to apply syntactic normalizations.

FN_4 The same problem with *PS* causes two similar functions to be missed, where one parameter was renamed from `rgb` to `rgba`.

FN_5 One false negative was found in files with the same name and these files were in fact highly similar. However, they were also very regular and *PS* was only 0.7386, that is, below the Gnome threshold for *PS*. This difference was caused by some imprecision in the oracle. The clone candidate consists of two parts, the first one being identical and the second different in terms of the parameters (of course not in terms of the other tokens as it is a type-2 clone). After the differing part, another segment follows that is identical to the second part of the first fragment. That is, one file seems to

DOI: 10.1002/smr

be an extension of the other file. Because the first part was identical, the author decided to accept it as a real clone, although he disagreed on the second part. By increasing the level of granularity from fragments to whole files, one could handle such cases. Then similar files could be presented as a whole.

FN_6 False negative FN_6 is an interesting case. The similar code is a list of `#define` statements defining values for MD5 calculation. One fragment is shown here:

```
#define T1_0  0xd76aa478
#define T1_1  0xe8c7b756
#define T1_2  0x242070db
#define T1_3  0xc1bdceee
#define T1_4  0xf57c0faf
#define T1_5  0x4787c62a
...
```

The values are all the same, but the macro names differ. However, the names are very similar to each other. While `T1_0` was used in one fragment, `T1` was used in the other one. And where `T1_1` was used in the first fragment, the second one uses `T2`. That is, the naming scheme is very similar, but our parameter similarity does not analyze the token values. They are either the same or different. Comparing the token values of parameters using string similarity could help here. Yet, this comparison would need to recognize the different enumeration offsets. While one fragment starts its naming with 0, the other one starts with 1.

FP_1 One false positive is a GTK marshalling code, whose high similarity is imposed by the GTK interface. One of the fragments is shown here (layout adjusted to save space):

```
gpointer arg_2, guint arg_3, guint arg_4, guint arg_5, gpointer data2);
register GMarshalFunc_VOID__UINT_BOXED_UINT_FLAGS_FLAGS callback;
register GCClosure *cc = (GCClosure *) closure;
register gpointer data1, data2;

g_return_if_fail (n_param_values == 6);
if (G_CCLOSURE_SWAP_DATA (closure))
{
   data1 = closure->data;
   data2 = g_value_peek_pointer (param_values + 0);
}
else
{
   data1 = g_value_peek_pointer (param_values + 0);
   data2 = closure->data;
}
callback =
  (GMarshalFunc_VOID__UINT_BOXED_UINT_FLAGS_FLAGS) (marshal_data ?
                                                    marshal_data : cc->
                                                    callback);
callback (data1,
          g_marshal_value_peek_uint (param_values + 1),
          g_marshal_value_peek_boxed (param_values + 2),
          g_marshal_value_peek_uint (param_values + 3),
          g_marshal_value_peek_flags (param_values + 4),
          g_marshal_value_peek_flags ...
```

It shares some identical code, but has differences in parameters in other places. The author did not consider it an interesting case for license violation assessment because the similarity is enforced by the library code used. This particular example pinpoints to a general problem. For instance, GUI code often tends to be very regular, too, because there is little variation in how widgets are created and configured. One way to tackle this problem would be to use more restrictive criteria for similarity in calls to particular libraries.

DOI: 10.1002/smr

FP_2 Another false positive is a list of array values, where most of these values are 0. Although these array fragments are highly similar in terms of *PS*, the similarity is just accidental. Similar array values may sometimes indicate copy violations, for instance, if certain codes are copied, however, the value 0 generally carries little value. One could treat them specially in defining similarity.

FP_3 Another false positives is similar to FP_2. This time, the macro `NULL` occurred consistently with a type conversion to `iw_handler` in front of, as shown here:

```
  return 0;
}
#endif
static const iw_handler rt_handler[] = {
  (iw_handler) NULL,            /* SIOCSIWCOMMIT */
  (iw_handler) NULL,            /* SIOCGIWNAME  1 */
  (iw_handler) NULL,            /* SIOCSIWNWID */
  (iw_handler) NULL,            /* SIOCGIWNWID */
  (iw_handler) NULL,            /* SIOCSIWFREQ */
  (iw_handler) NULL,            /* SIOCGIWFREQ 5 */
  (iw_handler) NULL,            /* SIOCSIWMODE */
  (iw_handler) NULL,            /* SIOCGIWMODE */
  (iw_handler) NULL,            /* SIOCSIWSENS */
  (iw_handler) NULL,            /* SIOCGIWSENS */
  (iw_handler) NULL /* not used */ ,    /* SIOCSIWRANGE */
  (iw_handler) rt_ioctl_giwrange,       /* SIOCGIWRANGE         11 */
  (iw_handler) NULL /* not used */ ,    /* SIOCSIWPRIV */
  (iw_handler) NULL /* kernel code */ , /* SIOCGIWPRIV */
  (iw_handler) NULL /* not used */ ,    /* SIOCSIWSTATS */
  (iw_handler) NULL /* kernel code */ , /* SIOCGIWSTATS         f */
  (iw_handler) NULL,            /* SIOCSIWSPY */
  (iw_handler) NULL,            /* SIOCGIWSPY */
  (iw_handler) NULL,            /* -- hole -- */
  (iw_handler) NULL,            /* -- hole -- */
  (iw_handler) NULL,            /* SIOCSIWAP */
  (iw_handler) NULL,            /* SIOCGIWAP    0x15 */
  (iw_handler) NULL,            /* -- hole --   0x16 */
```

Both fragments define some arrays for signal handlers. They are almost identical, but this similarity is imposed by the way signal handler arrays are declared.

FP_4 One false positive was even a type-1 clone. However, the parameters were named identical only by accident, because the terms were relatively meaningless `args[0]`, `args[1]`, and so on. The fragment is shown here:

```
args[0], args[1], args[2], args[3], args[4], args[5],
args[6], args[7], args[8], args[9], args[10], args[11],
args[12], args[13], args[14], args[15], args[16], args[17],
args[18], args[19], args[20]);
```

It is difficult to detect such cases. One would need to have an operational definition of a meaningless identifier.

4.4. Threats to Validity

The interpretation of our results is limited by several factors. We analyzed only C programs in this study. We also reported results for a very large Java corpus in our conference paper [19], but we did not consider any other programming language beyond these two.

The Ubuntu corpus is relatively large and diverse but certainly covers not all C programs that are written or may be written. More critical might be the choice of the subject systems. The only guideline used for the selection was to obtain a set of subject systems of increasing size. Other than that their selection was random. We had one outlier in the performance study, hinting at the possibility that there may be other systems that uncover different runtime characteristics.

For the study on using automated machine learning to derive a decision model to exclude spurious clone candidates, we used a family of Gnome projects. To mitigate the influence of the actual sample

DOI: 10.1002/smr

chosen for calibrating the parameters by the machine learner, we used 10-fold cross-validation. Yet, our results depend upon the chosen projects and the Ubuntu corpus and may not necessarily generalize to other systems and other corpora. Also, it was our own subjective judgment whether a clone would be relevant. Our study should be extended to additional systems and involve other independent human oracles.

5. CONCLUSION

In this paper, we extended our suffix-tree based inter-system clone detection by using a filter based on hashes for n-grams. Our performance study for a very large corpus has shown that suffix-tree based inter-system clone detection is already very fast and scales well to very large corpora. The study also showed that it can be made even faster by using a hashing filter when the same corpus is compared more than once. If the analysis is run only once, it is cheaper to avoid the hash index. This index-based filtering would support a service provider who offers license-violation detection as a service through the Internet, for instance. The cost of creating the index then amortizes for multiple searches.

To improve precision, the approach uses filtering based on metrics and automated machine learning to learn a decision model from a validated sample. Our evaluation shows that precision can in fact be improved substantially using relatively simple metric-based filters. We also found that a decision model derived from one sample may in fact be transferred to other systems with high recall and precision.

The analysis of the false negatives shows that our definition of parameter similarity must be improved. Four out of six false negatives could have been found. Looking at the similarity of different names would be another point of improvement for better recall. Also, some parameters' token values could be handled specially to increase precision further, for instance, 0 carries less information than other integer values. Also, the idea of excluding certain patterns imposed by libraries may help to reduce the number of false positives.

The vast majority of clone candidates could be excluded by the decision model, so that the analyst needs to analyze only a small fraction of the clone candidates. In particular, parameter similarity turned out to be a robust and effective filter. The decision models for both samples showed a similar threshold for it. Yet, we need to be careful here. Modern integrated development environments provide simple renaming refactorings, offering a malicious programmer a simple way of obfuscation that has no negative effect on maintainability. For this reason, we should not abstract from the positioning of identifiers. Using consistency of parameter substitution is also a way to improve recall as we have seen in our study.

Currently, our approach finds only type-1 and type-2 clones. We believe that this is sufficient for the use case of license-violation detection 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, if the goal is to do refactoring or detect re-useable code for libraries, for instance, summarizing type-1 and type-2 clones into larger type-3 clones might be beneficial. We plan to extend our approach to type-3 clones. On the technical side, using suffix arrays rather than suffix trees might be another way to improve performance. Suffix arrays are faster to create and are also much smaller in size.

ACKNOWLEDGEMENT

I would like to thank 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

DOI: 10.1002/smr

1. Yamashina T, Uwano H, Fushida K, Kamei Y, Nagura M, Kawaguchi S, Iida H. Shinobi: A real-time code clone detection tool for software maintenance. *Working Conference on Reverse Engineering*, IEEE Computer Society Press, 2009; 313–314.
2. Bazrafshan S, Koschke R, Göde N. Approximate code search in program histories. *Working Conference on Reverse Engineering*, IEEE Computer Society Press, 2011; 109–118.
3. Keivanloo I, Rilling J, Charland P. Seclone – a hybrid approach to internet-scale real-time code clone search. *International Conference on Program Comprehension*, IEEE Computer Society Press, 2011; 223–224.
4. Keivanloo I, Rilling J, Charland P. Internet-scale real-time code clone search via multi-level indexing. *Working Conference on Reverse Engineering*, IEEE Computer Society Press, 2011; 23–27.
5. Barbour L, Yuan H, Zou Y. A technique for just-in-time clone detection in large scale systems. *International Conference on Program Comprehension*, IEEE Computer Society Press, 2010; 76–79.
6. Ekwa Duala-Ekoko MR. Clonetracker: Tool support for code clone management. *International Conference on Software Engineering*, ACM Press, 2008; 843–846.
7. de Wit M, Zaidman A, van Deursen A. Managing code clones using dynamic change tracking and resolution. *International Conference on Software Maintenance*, IEEE Computer Society Press, 2009; 169–178.
8. Johnson JH. Identifying redundancy in source code using fingerprints. *Conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1993; 171–183.
9. Baker BS. On Finding Duplication and Near-Duplication in Large Software Systems. *Working Conference on Reverse Engineering*, IEEE Computer Society Press, 1995; 86–95.
10. Mayrand J, Leblanc C, Merlo EM. Experiment on the Automatic Detection of Function Clones in a Software System using Metrics. *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press: Washington, 1996; 244–254.
11. Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone Detection Using Abstract Syntax Trees. *International Conference on Software Maintenance*, Koshgoftaar TM, Bennett K (eds.), IEEE Computer Society Press, 1998; 368–378.
12. Roy C, Cordy J. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. *International Conference on Program Comprehension*, IEEE Computer Society Press, 2008; 172–181.
13. Hummel B, Juergens E, Heinemann L, Conradt M. Index-based code clone detection: Incremental, distributed, scalable. *International Conference on Software Maintenance*, IEEE Computer Society Press, 2010; 1–9.
14. Mende T, Koschke R, Beckwermert F. An evaluation of code similarity identification for the grow-and-prune model. *Journal on Software Maintenance and Evolution* March–April 2009; **21**(2):143 – 169.
15. Mende T, Beckwermert F, Koschke R, Meier G. Supporting the grow-and-prune model in software product lines evolution using clone detection. *European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, 2008; 163–172.
16. Higo Y, Tanaka K, Kusumoto S. Toward identifying inter-project clone sets for building useful libraries. *International Workshop on Software Clones*, ACM Press, 2010.
17. Livieri S, German DM, Inoue K. A needle in the stack: efficient clone detection for huge collections of source code. *Technical report*, University of Osaka Jan 2010.
18. Cordy JR. Exploring large-scale system similarity using incremental clone detection and live scatterplots. *International Conference on Program Comprehension*, IEEE Computer Society Press, 2011; 151–160.
19. Koschke R. Large-scale inter-system clone detection using suffix trees. *European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, 2012; 309–318.
20. Koschke R. Survey of research on software clones. *Duplication, Redundancy, and Similarity in Software*, Koschke R, Merlo E, Walenstein A (eds.), no. 06301 in Dagstuhl Seminar Proceedings, Dagstuhl: Dagstuhl, Germany, 2007. URL http://drops.dagstuhl.de/opus/volltexte/2007.
21. Koschke R. Frontiers in software clone management. *Frontiers in Software Maintenance, Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, 2008; 119–128.
22. Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Journal of Science of Computer Programming* 2009; **74**(7):470–495, doi:doi:10.1016/j.scico.2009.02.007. Special Issue on Program Comprehension (ICPC 2008).
23. Baker BS. A program for identifying duplicated code. *Computer Science and Statistics 24: Proceedings of the 24th Symposium on the Interface*, 1992; 49–57.
24. Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* Sep 2007; **33**(9):577–591.
25. Kamiya T, Kusumoto S, Inoue K. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
26. Koschke R, Falke R, Frenzel P. Clone detection using abstract syntax suffix trees. *Working Conference on Reverse Engineering*, IEEE Computer Society Press, 2006; 253–262.
27. Falke R, Koschke R, Frenzel P. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering* 2008; **13**(6):601–643, doi:10.1007/s10664-008-9073-9.
28. Livieri S, Higo Y, Matushita M, Inoue K. Very-large scale code clone analysis and visualization of open source. *International Conference on Software Engineering*, ACM Press, 2007; 106–115.
29. McCreight E. A space-economical suffix tree construction algorithm. *Journal of the ACM* 1976; **32**(2):262–272.
30. Ukkonen E. On-line construction of suffix trees. *Algorithmica 14* 1995; :249–260.
31. Chang WI, Lawler EL. Sublinear expected time approximate string matching and biological applications. *Algorithmica* 1994; **12**:327–344.
32. Gusfield D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 2008.
33. Juergens E, Deissenboeck F, Hummel B. Clonedetective - a workbench for clone detection research. *International Conference on Software Engineering*, ACM Press, 2009.

DOI: 10.1002/smr

34. Higo Y, Kamiya T, Kusumoto S, Inoue K. Method and implementation for investigating code clones in a software system. *Information and Software Technology* 2007; **49**(9–10):985–998.
35. Tiarks R, Koschke R, Falke R. An assessment of type-3 clones as detected by state-of-the-art tools. *International Workshop on Source Code Analysis and Manipulation*, IEEE Computer Society Press, 2009; 67–76.
36. Tiarks R, Koschke R, Falke R. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal* 2011; **19**(2):295–331.
37. Falke R. Erkennung von Falsch-Positiven Softwareklonen mittels Lernverfahren. Phd dissertation, University of Bremen, Germany 2011.
38. Puglisi SJ, Smyth WF, Yusufu M. Fast optimal algorithms for computing all the repeats in a string. *PSC*, 2008; 161–169.
39. Mertzel NJ. Copying 0.03 percent of software code base not "de minimis". *Journal of Intellectual Property Law & Practice* 2008; **9**(3):547–548.
40. Ueda Y, Kamiya T, Kusumoto S, Inouie K. On detection of gapped code clones using gap locations. *Proc. of Asia-Pacific Software Engineering Conference*, IEEE Computer Society Press, 2002; 327–336.
41. Jia Y, Binkley D, Harman M, Krinke J, Matsushita M. Kclone: A proposed approach to fast precise code clone detection. *International Workshop on Software Clones*, 2009.
42. Göde N. Clone evolution. PhD dissertation, University of Bremen, Bremen, Germany 2011.
43. Krinke J, Gold N, Jia Y, Binkley D. Cloning and copying between gnome projects. *Working Conference on Mining Software Repositories (MSR)*, IEEE Computer Society Press, 2010; 98–101.
44. Mende T. On the evaluation of defect prediction models. Phd dissertation, University of Bremen, Germany 2011.

AUTHORS' BIOGRAPHIES

Rainer Koschke is a full professor for software engineering at the University of Bremen in Germany and head of the software engineering group. He holds a doctoral degree in computer science from the University of Stuttgart, Germany, and is the current Chair of the IEEE TCSE committee on reverse engineering. His research interests are primarily in the fields of software engineering and program analyses. His current research includes architecture recovery, feature location, program analyses, clone detection, and reverse engineering. He is one of the founders of the Bauhaus research project (founded in 1997) and its spin-off Axivion GmbH (founded in 2006) to develop methods and tools to support software maintainers in their daily job.

DOI: 10.1002/smr