DECKARD: Scalable and Accurate Tree-based Detection of Code Clones*

Lingxiao Jiang Ghassan Misherghi Zhendong Su
University of California, Davis
{lxjiang,ghassanm,su}@ucdavis.edu

Stéphane Glondu
ENS de Cachan, France
steph@glondu.net

Abstract

Detecting code clones has many software engineering applications. Existing approaches either do not scale to large code bases or are not robust against minor code modifications. In this paper, we present an efficient algorithm for identifying similar subtrees and apply it to tree representations of source code. Our algorithm is based on a novel characterization of subtrees with numerical vectors in the Euclidean space $\mathbb{R}^n$ and an efficient algorithm to cluster these vectors w.r.t. the Euclidean distance metric. Subtrees with vectors in one cluster are considered similar. We have implemented our tree similarity algorithm as a clone detection tool called DECKARD *and evaluated it on large code bases written in C and Java including the Linux kernel and JDK. Our experiments show that* DECKARD *is both scalable and accurate. It is also language independent, applicable to any language with a formally specified grammar.*

1. Introduction

Many software engineering tasks, such as refactoring, understanding code quality, or detecting bugs, require the extraction of syntactically or semantically similar code fragments (usually referred to as "clones"). Various studies show that much duplicated code exists in large code bases [10, 11, 17]. Many such duplications can be attributed to poor programming practice since programmers often copy-paste code to quickly duplicate functionality. This tendency not only produces code that is difficult to maintain, but may also introduce subtle errors [6, 17].

Different approaches for clone detection have been proposed in the literature. Most of them focus on detecting syntactic similarity of code because checking semantic similarity is very difficult (and in general undecidable). Roughly, these techniques can be classified into four categories:

*This research was supported in part by NSF NeTS-NBD Grant No. 0520320, NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

String-based: A program is first divided into strings, usually lines. Each code fragment consists of a contiguous sequence of strings. Two code fragments are similar if their constituent strings match. The representative work here is Baker's "parameterized" matching algorithm [1, 2], where identifiers and literals are replaced with a global constant.

Token-based: A program is lexed to produce a token sequence, which is scanned for duplicated token subsequences that indicate potential code clones. Compared to string-based approaches, a token-based approach is usually more robust against code changes such as formatting and spacing. CCFinder [10] and CP-Miner [17] are perhaps the most well-known among token-based techniques.

Tree-based: A program is parsed to produce a parse tree or abstract syntax tree (AST) representation of the source program. Exact or close matches of subtrees can then be identified by comparing subtrees within the generated parse tree or AST [4, 5, 21]. Alternatively, different metrics can be used to *fingerprint* the subtrees, and subtrees with similar fingerprints are reported as possible duplicates [15, 19].

Semantics-based: Semantics-aware approaches have also been proposed. Komondoor and Horwitz [14] suggest the use of program dependence graphs (PDGs) [8] and program slicing [22] to find isomorphic PDG subgraphs in order to identify code clones. They also propose an approach to group identified clones together while preserving the semantics of the original code [13] for automatic procedure extraction to support software refactoring. Such techniques have not scaled to large code bases.

Of existing techniques, CCFinder [10], CP-Miner [17], and CloneDR [4, 5] represent the state-of-the-art. However, they either have limited scalability or are not robust against code modifications. Our goal is to develop a practical detection algorithm that is both scalable and robust against code modifications.

In this paper, we introduce a novel algorithm for detecting similar trees and a practical implementation, DECKARD, for detecting code clones. The main idea of the algorithm is to compute certain *characteristic vectors* to approximate structural information within ASTs and then

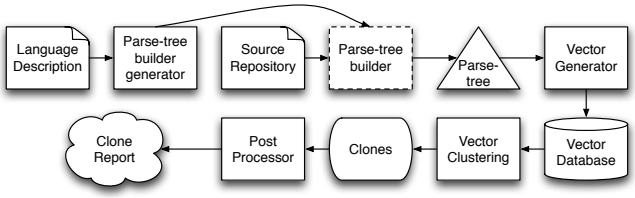


Figure 1. System architecture.

adapt *Locality Sensitive Hashing* (LSH) [7] to efficiently cluster similar vectors (and thus code clones).

Figure 1 shows the high level architecture of DECKARD: (1) A parser is automatically generated from a formal syntax grammar; (2) The parser translates sources files into parse trees; (3) The parse trees are processed to produce a set of vectors of fixed-dimension, capturing the syntactic information of parse trees; (4) The vectors are clustered w.r.t. their *Euclidean distances*; and (5) Additional post-processing heuristics are used to generate clone reports.

We have done extensive empirical evaluation of DECKARD on large software (including JDK and the Linux kernel) and compared it against CloneDR and CP-Miner. Results indicate that DECKARD is both scalable and accurate: it detects more clones in large code bases than both CloneDR and CP-Miner; it is more scalable than CloneDR, which is also tree-based, and is as scalable as the token-based CP-Miner.

The rest of the paper is structured as follows. We first give a detailed overview of our algorithm and illustrate it with an example (Section 2) before presenting details of our detection algorithm (Section 3). Next, we discuss our implementation and evaluation of DECKARD (Section 4). Finally, we survey related work (Section 5) and conclude with a discussion of future work (Section 6).

2. Overview

This section illustrates the main steps of our algorithm with a small example. Consider the following two C program fragments for array initialization:

```
for (int i= 0; i < n; i++)
    x[i]= 0;
```

```
for (int i= 0; i < n; i++)
    y[i]= "";
```

The parse trees for these two code segments are identical, because the code differs only in identifier names and literal values. The parse tree is shown in Figure 2. A pairwise tree comparison could be used to detect such clones, but this is expensive for large programs because of the possibly large number of subtrees. In the following, we demonstrate a novel, efficient technique for tree similarity detection.

Characteristic Vectors We introduce *characteristic vectors* to capture structural information of trees (and forests). This is a key step in our algorithm. The characteristic vector of a subtree is a point $\langle c_1, \ldots, c_n \rangle$ in the Euclidean space, where each c_i represents the count of occurrences of a specific tree pattern in the subtree. For this example, we let the tree patterns be the node kinds in a parse tree. We will introduce more general tree patterns in Section 3.2.1.

Not all nodes in parse trees are essential for capturing tree structural information; many are redundant w.r.t. their parents, or were introduced to simplify the grammar specification. We thus also distinguish between *relevant* and *irrelevant nodes*. Example irrelevant nodes include C tokens '[' and ']' and parentheses ('(' and ')'). In Figure 2, nodes with solid outlines are relevant while nodes with dotted outlines are irrelevant. Irrelevant nodes do not have an associated pattern or dimension in our vectors. For the example, the ordered dimensions of characteristic vectors are occurrence counts of the relevant nodes: id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, and for_s. Thus, the characteristic vector for the subtree rooted at decl is $\langle 1, 1, 0, 0, 0, 0, 0, 1, 0 \rangle$ because there is an id node, a lit node, and a decl node.

Characteristic vectors are generated with a post-order traversal of the parse tree by summing up the vectors for children with the vector for the parent's node. As an example, the vector for the subtree rooted at assign_e $\langle 2, 1, 1, 0, 1, 0, 0, 0, 0 \rangle$ is the sum of the vectors for array_e ($\langle 2, 0, 0, 0, 1, 0, 0, 0, 0 \rangle$), = ($\langle 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$), primary_e ($\langle 0, 1, 0, 0, 0, 0, 0, 0, 0 \rangle$), and the additional node assign_e ($\langle 0, 0, 1, 0, 0, 0, 0, 0, 0 \rangle$). Users may also specify a minimum token count to suppress vectors for small subtrees; this helps to avoid reporting small clones which are often uninteresting. For example, in Figure 2, with this threshold set to three, no vector is generated for the subtree rooted at incr_e. By varying this threshold, we can systematically find only large clones.

Vector Merging The aforementioned technique considers only those code fragments with a corresponding subtree in the parse tree. However, developers often insert code fragments within some larger context. Differences in the surrounding nodes may prevent the parents from being detected as clones (see Section 4.3.2 for a concrete example from JDK 1.4.2). To identify these cloned fragments, we use a second phase of characteristic vector generation, called *vector merging*, to sum up the vectors of certain node sequences. In this phase, a sliding window moves along a serialized form of the parse tree. The windows are chosen so that a *merged vector* contains a large enough code fragment. In Figure 2, for example, we merged the vectors for decl and cond_e to get the vector $\langle 3, 1, 0, 0, 0, 1, 0, 1, 0 \rangle$ for the combined code fragment.

The choice of which nodes in the tree to merge is important; these nodes must make good boundaries among cloned code, while not frequently containing large subtrees. Roots of expression trees, likely the atomic units for copy-pasting, are usually good choices for merging vectors. We call such chosen nodes *mergeable* nodes. In Figure 2, the mergeable nodes are the four children of the for statement. It is not

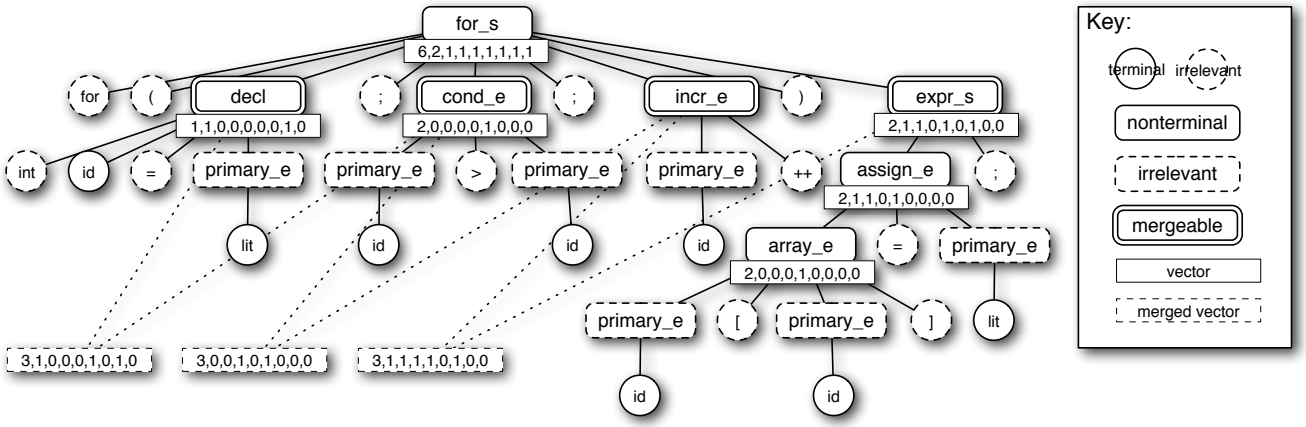


Figure 2. A sample parse tree with generated characteristic vectors.

necessary for mergeable nodes to be on a same level. If we had chosen any statement to be mergeable, the entire **for** loop would have been considered as one unit without subsequences. In Figure 2, we also required each merged fragment to contain at least five tokens. If we had required six tokens instead, there would have been only two merged vectors instead of three: (1) for decl and cond_e, and (2) for cond_e, incr_e, and expr_s.

Vector Clustering and Post-Processing After we have selected the characteristic vectors, our algorithm clusters similar characteristic vectors w.r.t. their Euclidean distances to detect cloned code. The two sample C code fragments both have the same characteristic vector $\langle 6, 2, 1, 1, 1, 1, 1, 1, 1 \rangle$, and DECKARD reports them as clones. Because the number of generated vectors can be large, an efficient clustering algorithm is needed. We will present such an algorithm in Section 3.

The subtree rooted at expr_s also illustrates the need for post-processing. When a particular subtree has a low branching factor, the vectors for a child and its parent are usually very similar and thus likely to be detected as clones. We employ a post-processing phase following clustering to filter such spurious clones.

3. Algorithm Description

In this section, we give a detailed technical description of our tree similarity algorithm: we first formally define a *clone pair* (Section 3.1), then introduce characteristic vectors for trees and describe how to generate them (Section 3.2), and finally explain our vector clustering algorithm for clone detection (Section 3.3).

3.1. Formal Definitions

In this paper, we view clones as syntactically similar code fragments. Thus, it is natural to define the notion of similar trees first. We follow the standard definition and use tree editing distance as the measure for tree similarity.

Definition 3.1 (Editing Distance) The *editing distance* of two trees T_1 and T_2, denoted by $\delta(T_1, T_2)$, is the *minimal sequence* of edit operations (either relabel a node, insert a node, or delete a node) that transforms T_1 to T_2.

Definition 3.2 (Tree Similarity) Two trees T_1 and T_2 are *σ-similar* for a given threshold σ, if $\delta(T_1, T_2) < \sigma$.

We are now ready to define the notion of a *clone pair*.

Definition 3.3 (Clone Pair) Two code fragments C_1 and C_2 are called a *clone pair* if their corresponding tree representations T_1 and T_2 are σ-similar for a specified σ.

Such a definition based on tree editing distance faithfully captures how similar two code fragments are. However, it does not lead naturally to an efficient algorithm because: (1) the complexity of computing the editing distance between two trees is expensive,[1] and (2) it requires many pairwise comparisons to locate similar code in large software (quadratic in the worst case). Instead, we approximate tree structures using numerical vectors and reduce the tree similarity problem to detecting similar vectors. Before describing the details, we define the two common distance measures for numerical vectors that we use in this paper.

Definition 3.4 (Distance Measures on Vectors) Let $v_1 = \langle x_1, \ldots, x_n \rangle$ and $v_2 = \langle y_1, \ldots, y_n \rangle$ be two n-dimensional vectors. The *Hamming distance* of v_1 and v_2, $\mathcal{H}(v_1, v_2)$, is their l_1 norm, *i.e.*, $\mathcal{H}(v_1, v_2) = ||v_1 - v_2||_1 = \sum_{i=1}^{n} |x_i - y_i|$. The *Euclidean distance* of v_1 and v_2, $\mathcal{D}(v_1, v_2)$, is their l_2 norm, *i.e.*, $\mathcal{D}(v_1, v_2) = ||v_1 - v_2||_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$.

Such distance measures are much easier to compute and efficient algorithms for near-neighbor queries exist for numerical vectors. Based on these observations, we show how to abstract trees into vectors and how to efficiently cluster similar vectors to detect code clones.

[1] More precisely, for two trees T_1 and T_2 the complexity is $O(|T_1| \times |T_2| \times d_1 \times d_2)$, where $|T_i|$ is the size of T_i and d_i is the minimum of the depth of T_i and the number of leaves of T_i [24].

3.2. Characteristic Vectors for Trees

Recall that in Section 2 we illustrated the use of occurrence counts of relevant nodes to abstract a subtree (or subtrees). That example shows a special case of the general construction that we will introduce in this section. In particular, we describe a general technique to map a tree (or forests) to a numerical vector which characterizes the structure of the given tree. Without loss of generality, we assume trees are binary [12].

3.2.1 Atomic Tree Patterns and Vectors

Given a binary tree, we define a family of *atomic tree patterns* to capture structural information of a tree. They are parametrized by a parameter q, the height of the patterns.

Definition 3.5 (q-Level Atomic Tree Patterns) A *q-level atomic pattern* is a complete binary tree of height q. Given a label set $\mathcal{L}$, including the empty label ϵ, there are at most $|\mathcal{L}|^{2^q-1}$ distinct q-level atomic patterns.

Definition 3.6 (q-Level Characteristic Vectors) Given a tree T, its *q-level characteristic vector* (denoted by $v_q(T)$) is $\langle b_1, b_2, \ldots, b_{|\mathcal{L}|^{2^q-1}}\rangle$, where b_i is the number of occurrences of the i-th q-level atomic pattern in T.

For example, in Figure 2, we used the relevant nodes as the 1-level atomic patterns and characterized trees with their 1-level characteristic vectors.

Abstracting trees as q-level vectors yields an alternative to the standard tree similarity definition based on editing distance. Our plan is to use Euclidean distance between q-level vectors to approximate the editing distance of the corresponding trees. We adapt a result of Yang *et al.* on computing tree similarity [23] to show that this approximation is accurate.

Theorem 3.7 (Yang *et al.*, Thm. 3.3 [23]) For any trees T_1 and T_2 with editing distance $\delta(T_1, T_2) = k$, the l_1 norm of the q-level vectors for T_1 and T_2, $\mathcal{H}(v_q(T_1), v_q(T_2))$, is no more than $(4q-3)k$.

For any two integer vectors v_1 and v_2, $\sqrt{\mathcal{H}(v_1, v_2)} \leq \mathcal{D}(v_1, v_2) \leq \mathcal{H}(v_1, v_2)$. Thus we have the following corollary that relates the tree editing distance of two trees with the Euclidean distance of their q-level vectors.

Corollary 3.8 For any trees T_1 and T_2 with editing distance $\delta(T_1, T_2) = k$, the l_2 norm of the q-level vectors for T_1 and T_2, $\mathcal{D}(v_q(T_1), v_q(T_2))$, is no more than $(4q-3)k$ and no less than the square root of the l_1 norm, *i.e.*,

$$\sqrt{\mathcal{H}(v_q(T_1), v_q(T_2))} \leq \mathcal{D}(v_q(T_1), v_q(T_2)) \leq (4q-3)k.$$

Corollary 3.8 suggests that either $\frac{\mathcal{D}(v_q(T_1), v_q(T_2))}{4q-3}$ or $\frac{\sqrt{\mathcal{H}(v_q(T_1), v_q(T_2))}}{4q-3}$ can be used as a lower bound of the tree editing distance $\delta(T_1, T_2)$. When such a lower bound is larger than a specific threshold σ, T_1 and T_2 cannot be σ-similar and thus not a clone pair for the specified σ. On the other hand, when the lower bound is smaller than σ, $\delta(T_1, T_2)$ is likely to be less than σ too. Hence, we reduce the problem of tree similarity to the problem of detecting similar q-level vectors.

Algorithm 1 q-Level Vector Generation

```
1:  function QVG(T : tree, C : configuration): vectors
2:      V ← ∅
3:      Traverse T in post-order
4:      for all node N traversed do
5:          V_N ← Σ_{n ∈ children(N)} V_n
6:          if IsRelevant(N, C) then
7:              id ← IndexOf(N, C)
8:              V_N[id] ← V_N[id] + 1
9:          end if
10:         if IsSignificant(N, C) ⋀
11:           ContainsEnoughTokens(V_N, C) then
12:             V ← V ⋃ {V_N}
13:         end if
14:     end for
15:     return V
16: end function
```

Notice that Definition 3.6, Theorem 3.7, and Corollary 3.8 can be relaxed to work on tree forests (a collection of trees) as well because tree forests can be viewed as a tree by adding an additional root. This is important for dealing with code fragments that do not correspond to a single subtree in the parse tree (*cf.* Section 2).

3.2.2 Vector Generation

There are two phases of vector generation: one for subtrees and one for subtree forests (for generating merged vectors). Algorithm 1 shows how vectors are generated for subtrees. Given a parse tree T, we essentially perform a post-order traversal of T to generate vectors for its subtrees. Vectors for a subtree are summed up from its constituent subtrees (line 5). Certain tree patterns may not be important for a particular application, so we distinguish between relevant and irrelevant tree patterns (a concept that is similar to and generalizes relevant and irrelevant nodes from Section 2). If a pattern rooted at a particular node N is relevant (line 6), we look up its index in the vector space using `IndexOf` (line 7) and update the vector correspondingly (line 8).

We also allow vectors to be generated only for certain subtrees, for example those that are more likely to be units of clones, such as subtrees rooted at declarations, expressions and statements. Users can select those *significant* node kinds to generate q-level vectors (line 10). For example, if array_e in Figure 2 had been specified as *insignificant*, no vector would have been generated for it. In addition, we may want to ignore small subtrees that contain too few tokens (*cf.* incr_e in Figure 2). Users can define a minimal token requirement on the subtrees, which is enforced with `ContainsEnoughTokens` (line 11).

Algorithm 2 shows how vectors are generated for adja-

Algorithm 2 Vector Merging for Adjacent Tree Forests

```
1: function WVG(T : tree, C : configuration): vectors
2:     ST ← Serialize(T, C);   V ← ∅
3:     step ← 0;   front ← ST.head
4:     back ← NextNode(ST.head, C)
5:     repeat
6:         V_merged ← ∑_{n∈[front,back]} V_n
7:         while back ≠ ST.tail ⋀
8:           ¬ContainsEnoughTokens(V_merged, C) do
9:             back ← NextNode(back, C)
10:            V_merged ← ∑_{n∈[front,back]} V_n
11:        end while
12:        if RightStep(step, C) then
13:            V ← V ⋃ {V_merged}
14:        end if
15:        front ← NextNode(front, C)
16:        step ← step + 1
17:    until front = ST.tail
18:    return V
19: end function
```

cent subtree forests. It serializes the parse tree T in post-order, then moves a *sliding window* along the serialized tree to merge q-level vectors from nodes within the sliding window. Because it is not useful to include every node in the serialized tree, we select certain node kinds (called *mergeable nodes*) as the smallest tree units to be included (to make larger code fragments in the context of clone detection). For example, the significant nodes, decl, cond_e, incr_e, and expr_s in Figure 2 are specified as mergeable. Users can specify any suitable node kinds as mergeable for a particular application. If both a parent and a child are mergeable, we exclude the child in the sliding window for the benefit of selecting larger clones. This is implemented by NextNode in Algorithm 2 (line 9).

Users can also choose the width of the sliding window and how far it moves in each step, *i.e.*, its *stride*. Larger widths allow larger code fragments to be encoded together, and may help detect larger clones, while larger strides reduce the amount of overlapping among tree fragments, and may reduce the number of spurious clones. With sliding windows of different widths, our algorithm can generate vectors for code fragments of different sizes and provide a systematic technique to find similar code of any size.

3.3. Vector Clustering

Given a large set of vectors $\mathcal{V}$, quadratic pairwise comparisons are computationally infeasible for similarity detection. Instead, we can hash vectors with respect to the Euclidean distances among them, and then look for similar vectors by only comparing vectors with equal hash values.

Locality Sensitive Hashing (LSH) [7,9] is precisely what we need. It constructs a special family of hash functions that can hash two similar vectors to the same hash value with arbitrarily high probability and hash two distant vectors to the same hash value with arbitrarily low probability. It also helps efficiently find near-neighbors of a query vector. In the following, we provide some basic background on LSH, then discuss how it is applied for clone detection.

3.3.1 Locality Sensitive Hashing

Definition 3.9 ((p_1, p_2, r, c)-Sensitive Hashing) A family $\mathcal{F}$ of hash functions $h : \mathcal{V} \rightarrow \mathcal{U}$ is called (p_1, p_2, r, c)-*sensitive* ($c \geq 1$), if $\forall v_i, v_j \in \mathcal{V}$,

$$\begin{cases} \text{if } \mathcal{D}(v_i, v_j) < r & \text{then } Prob[h(v_i) = h(v_j)] > p_1 \\ \text{if } \mathcal{D}(v_i, v_j) > cr & \text{then } Prob[h(v_i) = h(v_j)] < p_2 \end{cases}$$

For example, Datar *et al.* have shown that the following family of hash functions, which map vectors to integers, is locality sensitive [7]:

$$\{h_{\alpha,b} : \mathbb{R}^d \rightarrow \mathbb{N} \mid h_{\alpha,b}(v) = \lfloor \frac{\alpha \cdot v + b}{w} \rfloor,\ w \in \mathbb{R}, b \in [0, w]\}$$

Definition 3.10 ((r, c)-Approximate Neighbor) Given a vector v, a vector set $\mathcal{V}$, a distance r, and $c \geq 1$, $\mathcal{U} = \{u \in \mathcal{V} \mid \mathcal{D}(v, u) \leq cr\}$ is called an *rcAN* set of v, and any $u \in \mathcal{U}$ is a (r, c)-*approximate neighbor* of v.

Given a vector set $\mathcal{V}$ of size n and a query vector v, LSH may establish hash tables for $\mathcal{V}$ and find v's rcAN set in $O(dn^{\rho} \log n)$ time and $O(n^{\rho+1} + dn)$ space, where d is the dimension of the vectors and $\rho = \log_{p_2} p_1 < \frac{1}{c}$ for $c \in [1, +\infty)$. As long as we feed r (the largest distance allowed between v and its neighbors) and p_1 (the minimal probability that two similar vectors have the same hash value) to LSH, it automatically computes other parameters that would give optimal running time of a query.

3.3.2 LSH-based Clone Detection

LSH's querying functionality can help find every vector's rcAN sets, which are needed for clone detection. Algorithm 3 describes the utilization of LSH: (1) All vectors are stored into LSH's hash tables (line 2), where r serves as the threshold σ defined in Definition 3.3; (2) A vector v is used as a query point to get an rcAN set (lines 3 and 4); (3) If the rcAN set only contains v itself, it means v has no neighbors within distance σ and should be deleted directly (line 8); (4) Otherwise, the rcAN set is treated as a clone class (lines 6 and 8). Such a process may query LSH n times in the worst case. Thus, our LSH-based clone detection takes $O(dn^{\rho+1} \log n)$ time, where d is the dimension of the vectors, *i.e.*, $|\mathcal{L}|^{2^q-1}$ in terms of q-level vectors, where $|\mathcal{L}|$ is the number of node kinds in a parse tree.

All the rcAN sets may contain potentially spurious clones (*cf.* Section 2) and are post-processed to generate clone reports. A filter is created to examine the line range of every clone in an rcAN set and remove any that is contained by or overlaps with others. A second filter is applied after the first one to remove rcAN sets that contain only one vector. Both filters run in linear time in the number of rcAN sets and quadratic time in the size of the sets.

Algorithm 3 LSH-based Clone Detection

```
1: function LSHCD(V : vectors, r : distance, p1 : prob): rcANs
2:     N ← ∅;   LSH(V, r, p1)
3:     repeat   pick a v ∈ V
4:         rcAN ← queryLSH(v)
5:         if |rcAN| > 1 then
6:             N ← N ⋃{rcAN \ ⋃_{n∈N} n}
7:         end if
8:         V ← V \ rcAN
9:     until V = ∅
10:    return PostProcessing(N)
11: end function
```

3.4. Size-Sensitive Clone Detection

Definition 3.3 of a clone pair does not take into account the varying sizes of code fragments. It is however natural to allow more edits for larger code fragments to be still considered clone pairs. In this section, we introduce a size-sensitive definition of code clones and an algorithm for detecting such clones. Such a higher tolerance to edits for larger code fragments facilitates the detection of more large clones.

Definition 3.11 (Code Size) The *size* of a code fragment C in a program P, denoted by $S(C)$, is the size of its corresponding tree fragments in the parse tree of P.

Definition 3.12 (Size-Sensitive Clone Pair) Two code fragments C_1 and C_2 form a *size-sensitive clone pair* if their corresponding tree representations T_1 and T_2 are $f(\sigma, S(C_1), S(C_2))$-similar, where f is a monotonic, non-decreasing function with respect to σ and $S(C_i)$.

Clone detection based on Definition 3.12 requires larger distance thresholds for larger code. We now present a technique to meet such a requirement. The basic idea is *vector grouping*: vectors for a program are separated into different groups based on the sizes of their corresponding code fragments; then LSH is applied on each group with an appropriate threshold; and finally, all reported clone classes from different groups are combined.

Any grouping strategy is appropriate as long as it meets the following requirements: (1) It should not miss any clones detectable with a fixed threshold, thus each group should overlap with the neighboring groups; (2) It should not produce many duplicate clones, thus overlapping should be avoided as much as possible; (3) It should produce many small groups to help reduce clustering cost.

Algorithm 4 shows a generic vector grouping algorithm, where s is a user-specified code size for the first group. Each vector v is dispatched into groups whose size ranges contain the size of its corresponding code fragment, *i.e.*, $S(C_v)$. SIZERANGES shows our formulae for grouping. The exact constraints used to deduce the grouping formulae can vary as long as they meet the aforementioned requirements.

Algorithm 4 Vector Grouping

```
1: function VG(V : vectors, r : distance, s : size)
2:     R ← sizeRanges(V, r, s)
3:     dispatch V into groups according to the ranges in R
4: end function
5:
6: function SIZERANGES(V : vectors, r : distance, s : size)
7:     The code size range for the 1st group ← [0, s + r]
8:     The range for the 2nd group ←
9:                         r = 0 ? [s+1, s+1] : [s, s+3r+1]
10:    repeat   compute [l_{i+1}, u_{i+1}] as
11:        l_{i+1} ← r = 0 ? (u_i + 1) : (u_i − (l_i/s) r)
12:        u_{i+1} ← r = 0 ? (u_i + 1) : ((s+2d)/s u_i − 2 d^2/s^2 l_i + 1)
13:    until u_i ≥ max_{v∈V}{S(C_v)}
14: end function
```

We can estimate $S(C)$ with the size of C's vector $v = \langle x_1, \ldots, x_n \rangle$, *i.e.*, $S(C) \approx S(v) = \sum_{i=1}^{n} x_i$. Although irrelevant nodes may cause $S(v) < S(C)$, this should have little impact on clone detection because each $S(C)$ is adjusted accordingly.

It is also worth mentioning that vector grouping has the added benefit to improve scalability of our detection algorithm. Because the vectors are separated into smaller groups, the number of vectors will usually not be a bottleneck for LSH, thus enabling the application of LSH on larger programs. In addition, because vector generation works on a file-by-file basis and the separated vectors are processed one group at a time, our algorithm can be easily parallelized.

4. Implementation and Empirical Evaluation

This section discusses our implementation of DECKARD and presents a detailed empirical evaluation of it against two state-of-the-art tools: CloneDR [4, 5] and CP-Miner [17].

4.1. Implementation

We have implemented our algorithm as a clone detection tool called DECKARD. In our implementation, we use 1-level vectors to capture tree structures. DECKARD is language independent and works on programs in any programming language that has a context-free grammar. It automatically generates a parse tree builder to build parse trees required by our algorithm. DECKARD takes a YACC grammar and generates a parse tree builder by replacing YACC actions in the grammar's production rules with tree building mechanisms. The generated parse tree builders also have high tolerance for syntactic errors. Thus, DECKARD is more applicable than other tree-based clone detection tools, even for languages with incomplete or inaccurate grammars. As an example, only 2 files out of $8,453$ in JDK 1.4.2 cannot be parsed by DECKARD, whereas 81 cannot be parsed by CloneDR.

Section 4.3 will show that DECKARD works effectively for both C and Java. In addition, YACC grammars are

available for many languages, often with the requisite error recovery to localize syntax problems. Thus, it should be straightforward to port DECKARD to other languages.

4.2. Experimental Setup

We performed extensive experiments on DECKARD, and the most detailed ones were on JDK 1.4.2 (8,534 `java` files, 2,418,767 LoC) and Linux kernel 2.6.16 (7,988 `C` files, 5,287,090 LoC).[2] We also compared DECKARD to both CloneDR [4, 5], a well-known AST-based clone detection tool for Java, and CP-Miner [17], a token-based tool for C.

To compare with CloneDR, we ran experiments on a workstation with a Xeon 2GHz processor and 1GB of RAM, with both Windows XP (for CloneDR) and Linux kernel 2.4.27 (for DECKARD). CloneDR has several parameters that may affect its clone detection rates, and we chose the most lenient values for all those parameters: (1) The minimal depth of a subtree to be considered a clone is set to two; (2) The minimal number of tree nodes a clone should contain is set to three; (3) The maximal number of parameters allowed when using parameterized macros to refactor clones is set to 65535; and (4) $Similarity$ is set to a value between 0.9 and 1.0, where CloneDR [5] defines $Similarity$ as the following:

$$Similarity(T_1, T_2) = \frac{2H}{2H + L + R} \qquad \text{(Eq. 1)}$$

where H is the number of shared nodes in trees T_1 and T_2, L is the number of different nodes in T_1, and R is the number of different nodes in T_2. This definition takes tree sizes into account, similar to our definition in Section 3.4. To make our comparisons fair despite the different configuration options in each, we compute DECKARD's threshold σ from $Similarity$ as follows. Suppose v_1 and v_2 are the 1-level vectors for T_1 and T_2 respectively. Because the l_1 norm of v_1 and v_2 can be approximated as $L + R$ and $l_2 \geq \sqrt{l_1}$ for integer vectors, we can transform a given $Similarity$ s to an approximate l_2 distance:

$$\begin{aligned} \mathcal{D}_s(v_1, v_2) &\geq \sqrt{\mathcal{H}(v_1, v_2)} \approx \sqrt{L + R} \\ &\overset{\{\text{Eq. 1}\}}{=} \sqrt{(1-s) \times (|T_1| + |T_2|)} \\ &\geq \sqrt{2(1-s) \times min(S(v_1), S(v_2))} \end{aligned}$$

Given a vector group $\mathcal{V}$, $\sqrt{2(1-s) \times min_{v \in \mathcal{V}} S(v)}$ can serve as the threshold σ used by DECKARD for the group. This is similar to Section 3.4, where we use vector sizes to approximate tree sizes. In Figures 3 and 4, we show $Similarity$ only, without showing the derived σ.

To compare with CP-Miner (available for Linux), we ran experiments on a workstation running Linux kernel 2.6.16 with an Intel Xeon 3GHz processor and 2GB of RAM. CP-Miner uses a different distance metric, called *gap*, which is the number of statement insertions, deletions, or modifications to transform one statement sequence to another. Such a parameter is invariant w.r.t. different code sizes.

[2] We have also done experiments on the following programs and obtained consistent results: GCC 3.3.6 (C), PostgreSQL 8.1.0 (C), Derby 10.0.2.1 (Java), and Apache 2.2.0 (C). Due to space limitations, we do not report the detailed data here.

4.3. Experimental Results

We have evaluated DECKARD in terms of the following: clone quantity (*i.e.*, number of detected clones), clone quality (*i.e.*, number of false clones), and its scalability. Our results indicate that DECKARD performs significantly better than both CloneDR and CP-Miner.

4.3.1 Clone Quantity

We measure clone quantity by the number of lines of code that are within detected clone pairs.

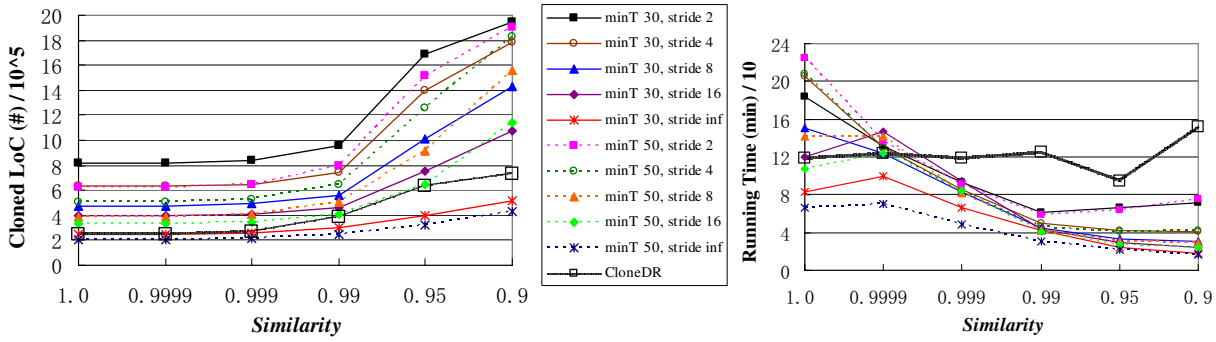
In the first experiment, we compared DECKARD with CloneDR on JDK. CloneDR failed to work on the entire JDK at once. It also failed on files with minor syntactic problems. Thus, we excluded those syntactically incorrect files reported by CloneDR and separated the remaining files into nine overlapping groups, with each group containing around 1,000 files. Figure 3(a) shows the total detected cloned lines over many runs on JDK. For DECKARD, we used a variety of configuration options: minT (minimal number of tokens required for clones) was set to 30 or 50, stride (size of the sliding window) ranged from 2 to inf (equivalent to no merging of vectors), and $Similarity$ ranged between 0.9 and 1.0. The setting with an infinite stride means that vector merging was disabled. The total number of cloned lines for DECKARD ranges from 204,263 to 1,943,777, while for CloneDR the number ranges from 246,708 to 727,701.

In our second experiment, we compared DECKARD with CP-Miner on the Linux kernel. Figure 4(a) shows the total number of detected clone lines by DECKARD under different configuration options with minT set to 30 or 50, stride ranging from 2 to inf, and $Similarity$ ranging from 0.9 to 1.0. The total number of detected cloned lines ranges from 338,519 to 3,936,242. For CP-Miner, we used four configuration options with minT set to 30 or 50 and gap set to 0 or 1. Its total number of detected clone lines ranges from 498,113 to 1,108,062 as shown in Table 1. It failed to operate with $gap > 1$.

In addition, Figure 4(c) plots the decline in clone detection rates as minT increases for both CP-Miner and DECKARD. Even with $Similarity$ set to 1.0, DECKARD detects more clones than CP-Miner.

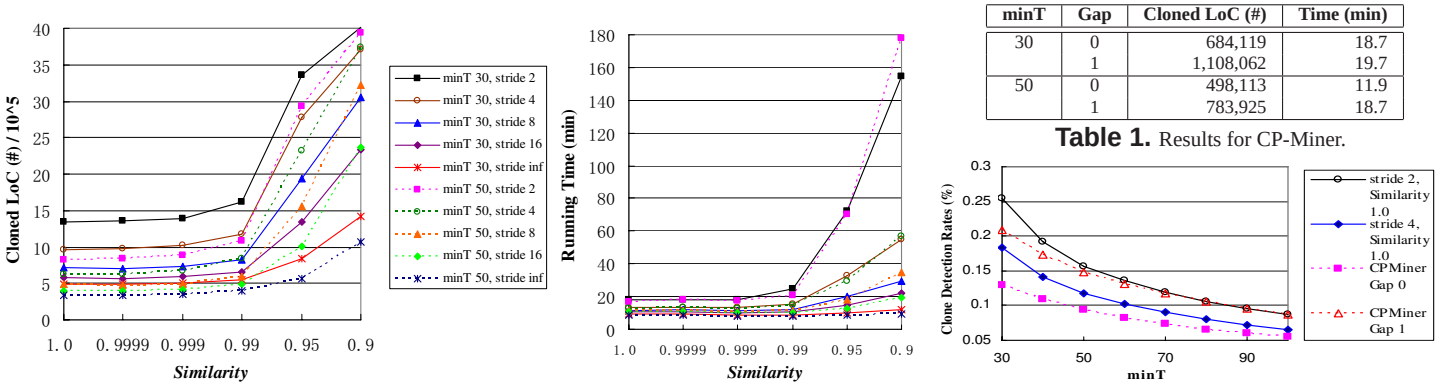
4.3.2 Clone Quality

The number of reported spurious clones is also important in assessing clone detection tools. We performed random, manual inspection on rcAN sets (*i.e.*, clustered similar vectors) using two criteria: (1) Does an rcAN set contain at least one clone pair that corresponds to copy-pasted fragments? (2) Are all clones in an rcAN set copies of one



(a) Cloned LoC (b) Running Time

Figure 3. Results for DECKARD (with grouping and full parameter tuning) and CloneDR on JDK 1.4.2.



(a) Cloned LoC (b) Running Time (c) Effects of minT on Clone Detection Rates

minT	Gap	Cloned LoC (#)	Time (min)
30	0	684,119	18.7
	1	1,108,062	19.7
50	0	498,113	11.9
	1	783,925	18.7

Table 1. Results for CP-Miner.

Figure 4. Results for DECKARD (with grouping and selective parameter tuning) and CP-Miner (Table 1) on Linux kernel 2.6.16.

another? If a set fails to satisfy either of the criteria, we classify it as a false clone report.

It may be difficult to decide for certain whether two code fragments are clones or not. For example, consider the following code fragments from JDK 1.4.2:

```
1  else if (option.equalsIgnoreCase("basic")) {
2          bBasicTraceOn = true;
3  } else if (option.equalsIgnoreCase("net")) {
4          bNetTraceOn = true;
5  } else if (option.equalsIgnoreCase("security")) {
6          bSecurityTraceOn = true;
7  } else ...
8  ...
9  else if (opt.equals("-nohelp")) {
10         nohelp = true;
11 } else if (opt.equals("-splitindex")) {
12         splitindex = true;
13 } else if (opt.equals("-noindex")) {
14         createindex = false;
15 } else ...
```

The code between lines 1–7 and that between lines 9–15 have identical structure but different variable names, functions, and constants. CloneDR and CP-Miner may detect them as clones if the two `if-else` sequences are standalone statements, but miss them if they are in the middle of different, larger `if-else` statements. DECKARD always detects them with reasonably small settings for minT and stride.

We inspected 100 randomly selected rcAN sets reported by DECKARD for JDK 1.4.2 with minT set to 50, stride set to 4, and *Similarity* set to 1.0. Of those, 93 rcAN sets are clearly real clones. Among the remaining seven rcAN sets, three involve `if-else` and `switch-case` that are similar to the above `if-else` example, three involve sequences of simple `import` statements, and one involves sequences of simple declarations. Although it is unclear whether these are clones, the reported clone pairs are all structurally the same. Also because both CloneDR and CP-Miner may detect such code as clones, we also classified these as real clones. This experiment indicates that DECKARD is highly accurate. Because the version of CloneDR that we have does not output the actual clones, we cannot directly compare its accuracy with DECKARD. For future work, we plan to develop a better user interface for DECKARD, which would allow us to conduct further user studies and to more rigorously assess the quality of reported clones.

4.3.3 Scalability

Table 2 shows the worst-case time and space complexities of CloneDR, CP-Miner, DECKARD, and LSH. Although the number of tree nodes n is usually several times larger than the number of statements m in a program, DECKARD's performance is still comparable to CP-Miner for large programs because ρ is usually much smaller than one. With vector grouping, LSH's memory consumption can be sig-

	CloneDR	CP-Miner	LSH	LSH w/ Grouping	DECKARD w/ Post-Processing
Time	$O(\frac{n^2}{\lvert Buckets\rvert})$	$O(m^2)$	$O(dn^\rho \log n)$	$O(d\sum_{g\in G}\lvert g\rvert^\rho \log\lvert g\rvert)$	$O(n + d\sum_{g\in G}\lvert g\rvert^{\rho+1}\log\lvert g\rvert + c\lvert rcAN\rvert^2)$
Mem	$O(n)$	$O(m)$	$O(n^{\rho+1} + dn)$	$O(max_{g\in G}\{\lvert g\rvert^{\rho+1} + d\lvert g\rvert\})$	$max\{O(c\lvert rcAN\rvert), O_{g\in G}(\lvert g\rvert^{\rho+1} + d\lvert g\rvert)\}$

Table 2. Worst-case complexities of CloneDR, CP-Miner, and DECKARD (m is the number of lines of code, n is the size of a parse tree, $|Buckets|$ is the number of hash tables used in CloneDR, d is the number of node kinds, $|g|$ is the size of a vector group, $0 < \rho < 1$, c is the number of clone classes reported, and $|rcAN|$ is the average size of the clone classes).

	Sim	G (#)	Cloned LoC (#)	T (min)
Full Tuning	1.0	1984	624265	224.8
Selective Tuning			624265	14.9
Full Tuning	0.99	235	792326	58.6
Selective Tuning			792298	16.3

Table 3. Effects of selective parameter tuning in LSH. The data is for JDK 1.4.2, with minT 50, stride 2.

nificantly reduced to make DECKARD scale to very large programs.

Figure 3(b) plots running times for both DECKARD and CloneDR on JDK. When $Similarity < 0.9999$, DECKARD is several times faster than CloneDR. We show next how DECKARD can be configured to run significantly faster. By default, LSH takes $O(kd\sum_{g\in G}|g|^\rho \log|g|)$ time to tune its own parameters and build optimal (w.r.t. query time) hash tables, where k is the number of iterations it uses to find the optimal parameters. Such cost accumulates when the vectors are split into groups, and thus LSH may spend much time on parameter tuning. Reusing the parameters computed for certain groups (*e.g.*, the largest group) can dramatically reduce LSH's running time with little effect on clone quantity and quality. Table 3 shows the effectiveness of such a strategy in reducing the overall running time of DECKARD, especially when the vectors are split into many groups.

Figure 4(b) shows DECKARD's running time on the Linux kernel with selective parameter tuning. When $Similarity > 0.95$, DECKARD runs in tens of minutes and is comparable to CP-Miner (*cf.* Table 1); it can be even faster when $Similarity$ is close to 1.0. When $Similarity \leq 0.95$, DECKARD may take more time than CP-Miner. This extra cost is reasonable considering that DECKARD is tree-based and detects more clones, while CP-Miner is token-based and cannot operate with $gap > 1$, and that $Similarity \leq 0.95$ is often too small for clone detection tasks.

5. Related Work

In this section, we discuss closely related work and split them into three categories: (1) tree similarity detection; (2) studies on code clones; and (3) clone detection algorithms.

Tree Similarity Detection Following the increased popularity of tree-structured data such as XML databases, similarity detection on trees is gaining increasing attention. However, efficient tree similarity detection still remains an open problem, while similarity detection on high dimension numerical vectors has already been extensively studied and efficient algorithms exist. Yang *et al.* [23] propose an approximation algorithm for computing tree editing distances. We adapt their characterization to capture structural information in parse trees, and apply LSH [7] to search for similar trees. To the best of our knowledge, DECKARD is the most effective and scalable tool for tree similarity detection.

Studies on Code Clones A few independent studies address the questions of clone coverage and evolution in large open-source projects. The goal for clone coverage is to determine what fraction of a program is duplicated code. It is difficult to directly compare these studies because such results are usually sensitive to: (1) the different definitions of code similarity used; (2) the particular detection algorithms used; (3) the various choices of parameters for these algorithms; and (4) the different code bases used for evaluation (*e.g.*, CCFinder [10] reports 29% cloned code in JDK, and CP-Miner [17] reports 22.7% cloned code in Linux kernel 2.6.6). However, these studies do confirm that there is a significant amount of duplicated code in large code bases.

The goal of clone evolution is to understand how clones are introduced or removed across different versions of a software. Laguë *et al.* [16] examined six versions of a telecommunication software system and found that a significant number of clones were removed due to refactoring, but the overall number of clones increased due to the faster rate of clone introduction. Kim *et al.* [11] describe a study of clone genealogies and find that: (1) many code clones are short-lived, so performing aggressive refactoring may not be worthwhile; and (2) long-lived clones pose great challenges to refactoring because they evolve independently and can deviate significantly from the original copy.

Clone Detection Many algorithms and tools exist for clone detection. First, there are tools specifically designed for estimating similarity in programs for the purpose of detecting plagiarism. Example tools include Moss [20] and JPlag (`http://www.jplag.de`). These tools are usually very coarse-grained and are not suitable for clone detection. Second, there are token-based tools, such as CP-Miner [17] and CCFinder [10]. These are usually efficient, scale to millions of lines of code, and find good quality clones, but they are sensitive to code restructuring and minor edits, so may miss clones. Third, there are tree-based techniques, which are less sensitive to code edits than token-based tools. Baxter *et al.* [4, 5] apply AST hashing for detecting exact and

near-miss clones. Wahler *et al.* [21] apply *frequent itemset* data mining techniques on ASTs represented in XML to detect clones with minor changes. DECKARD is also tree-based, but because of our novel use of characteristic vectors and efficient vector clustering techniques, it detects significantly more clones and is much more scalable. Finally, there are semantic-based techniques [14], which are most robust against code modifications, such as re-ordered statements, non-contiguous clones, and nested clones. However, these have not been shown to scale to large programs.

There is recent work applying clone detection algorithms to find "structural clones" for the purpose of detecting design-level similarities. For example, two different clone sets that often occur together in program files are an example of structural clones. Basit and Jarzabek [3] first apply CCFinder to detect *simple code clones* and then use a *frequent itemset* data mining algorithm to correlate simple clones to find design-level similarities. PR-Miner [18] also uses frequent itemset mining to detect implicit, high-level programming patterns for specification discovery or bug detection. Our algorithm can also be used for such purposes as long as we adjust vector generation to appropriately model these problems. We leave for future work the application of our algorithm on such pattern discovery tasks.

6. Conclusions and Future Work

In this paper, we have presented a practical algorithm for identifying similar subtrees and applied it to detect code clones. It is based on a novel characterization of trees as vectors in $\mathbb{R}^n$ that effectively captures structural information of trees and an efficient hashing and near-neighbor querying algorithm for numerical vectors. We have implemented our algorithm in the tool DECKARD. It is language independent and highly configurable. We have evaluated DECKARD on large code bases, including the Linux kernel and JDK. It easily scales to millions of lines of code and has identified more clones than existing tools. Our algorithm is general and can be extended to work on other data structures such as graphs. It also has many other potential applications, such as bug detection, code refactoring, and programming pattern discovery. For future work, we plan to apply our algorithm to such problem domains.

Acknowledgments

We thank Minya Dai for referring LSH to us and Alex Andoni for implementing LSH and providing us with its code. We also thank Zhenmin Li and Yuanyuan Zhou in the OPERA group at UIUC for providing a binary release of CP-Miner and Semantic Designs, Inc. for providing an evaluation version of CloneDR for use in our experiments. Finally, we thank Earl Barr, Chris Bird, Prem Devanbu, Ron Olsson, Gary Wassermann, Daniel Zinn, and the anonymous ICSE reviewers for useful feedback on earlier versions of this paper.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, 1995.

[2] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SICOMP*, 26(5):1343–1362, 1997.

[3] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE*, pages 156–165, 2005.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634, 2004.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[9] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

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

[11] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196, 2005.

[12] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Pub Co., 3rd edition, 1997.

[13] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL*, pages 155–169, 2000.

[14] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[15] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Soft. Eng.*, 3(1/2):77–108, 1996.

[16] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321, 1997.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[18] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, 2005.

[19] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–254, 1996.

[20] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD*, pages 76–85, 2003.

[21] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135, 2004.

[22] M. Weiser. Program slicing. *TSE*, 10(4):352–357, 1984.

[23] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, pages 754–765, 2005.

[24] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SICOMP*, 18(6):1245–1262, 1989.