

Plagiarism Detection across Programming Languages

Christian Arwin

S.M.M. Tahaghoghi

School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Melbourne 3001, Australia.
{carwin,saied}@cs.rmit.edu.au

Abstract

Plagiarism is a widespread problem in assessment tasks; in computing courses, students often plagiarise source code. For all but the smallest classes, manual detection of such plagiarism is impractical, and, while automated tools are available, none has been applied to detect *inter-lingual plagiarism*, where source code is copied from one language to another. In this work, we propose a novel approach, XPlag, to detect plagiarism involving multiple languages using intermediate program code produced by a compiler suite. We describe experiments to evaluate XPlag, and show that we can detect inter-lingual plagiarism with reasonably good precision.

Keywords: program source code similarity, plagiarism detection

1 Introduction

Plagiarism — the representation of another’s work as one’s own — is a serious problem for academics; a survey performed by Sheard, Dick, Markham, Macdonald & Walsh (2002) on a sample of students at Monash and Swinburne universities shows that 85.4% of 137 Monash University students and 69.3% of 150 Swinburne University students admitted to having engaged in academic dishonesty. Educational institutions commonly attempt to reduce the incidence of plagiarism by applying penalties for violation of rules on academic dishonesty, yet plagiarism remains widespread (Zobel & Hamilton 2002).

Many computing courses have assessment tasks that require submission of program source code; students may plagiarise by copying code from friends, the Web, or so-called “private tutors” (Zobel 2004). For large cohorts, manual comparison of submissions to identify plagiarism is impractical, and so students may feel confident that their work will escape detection. There are also commercial concerns; organisations may be unknowingly liable to litigation for unauthorised use of program source code. Robust co-derivative detection methods are essential.

Several approaches have been proposed for detecting code plagiarism; most use source code tokenisation and string matching algorithms to measure similarity. These generally perform well in detecting plagiarism that involves common disguising techniques such as statement reordering or modification of constant values and variable names (Gitchell

& Tran 1999, Prechelt, Malpohl & Philippsen 2000, Wise 1996).

However, these approaches are designed and applied to detect plagiarism between programs written in a single language, and cannot handle cases where source code is copied from one language to another. We propose the names *intra-lingual plagiarism* for the former, and *inter-lingual plagiarism* or *cross-lingual plagiarism* for the latter.

We hypothesise that plagiarised programs, regardless of the language they are written in, have a similar code structure. We propose two solutions to address inter-lingual plagiarism. One is to compare the tokens produced by most existing plagiarism detection approaches that support more than one language. The second is to compare the intermediate code produced by a *compiler suite*, that is, a compiler that supports more than one language. The former typically supports only a few languages and requires a scanner and parser for each language, while the latter solution relies on the components of an existing generic compiler suite. We focus on the second approach in this work.

We test our new approach against three collections using ground truth developed from an existing state-of-the-art plagiarism detection system and through manual comparisons. The results show that our approach detects plagiarism with reasonably good precision for all the test collections. More importantly, it succeeds in detecting plagiarism across languages.

The remainder of this thesis is organised as follows. In Section 2, we discuss the main computer-based approaches to plagiarism detection. In Section 3, we describe our approach for the detection of inter-lingual plagiarism. In Section 4, we describe our experiments and our analysis of the results. We conclude in Section 5 with a summary of our work and thoughts for future research.

2 Background

Two factors that complicate plagiarism detection are the abundance of available resources and the variety of techniques used to disguise the copied materials. A number of approaches have been proposed to detect plagiarism in text and in program source code; we briefly review some of these in this section.

2.1 Text Plagiarism Detection

Text plagiarism involves copying parts of manuscripts, papers, and documents. Hoad & Zobel (2003) explore the *ranking* and *fingerprinting* approaches for detecting plagiarism of text. These approaches have a common preprocessing stage that includes *case folding*, *stemming* (removing prefix/suffix from words), *stopping* (removing common words), and *term parsing* (removing whites-

pace, punctuation, and control characters from the document).

The ranking approach consists of two stages to find documents similar to a query. In the first stage, documents are indexed. In the second stage, terms in the query document are matched against the indexed terms of each collection document, and a similarity score is calculated. Documents are ranked by decreasing similarity score for presentation to the user. This approach relies on the use of an effective similarity function to determine the similarity score for each document (Hoad & Zobel 2003). The fingerprinting approach also uses the two stages used by the ranking approach. However, it compares document fingerprints rather than document terms.

2.2 Source Code Plagiarism Detection

The nature of program source code makes it difficult to apply simple text-based detection techniques. Copied code is typically altered to avoid detection. Whale (1986) lists thirteen techniques that students may use to disguise the origin of copied code; these are “changing comments, changing formatting, changing identifiers, changing the order of operands in expressions, changing data types, replacing expressions by equivalents, adding redundant statements, changing the order of time-independent statements, changing the structure of iteration statements, changing the structure of selection statements, replacing procedure calls by the procedure body, introducing non-structured statements, combining original and copied program fragments”. We consider there to be one additional item: the translation of source code from one language to another, or *inter-lingual plagiarism*. For example, source code written in C may be copied across to an implementation in Java.

There are several existing approaches to detect code plagiarism. Prechelt et al. (2000) identify two main categories of automated plagiarism detection for program source code; these are *feature comparison* and *structure comparison*. We explain these below.

2.2.1 Feature Comparison

In feature comparison, the similarity of two programs is estimated from the similarity of various software metrics, such as the average number of characters per line, the number of comment lines, the number of indented lines, the number of blank lines, and the number of tokens (for example, keywords, operator symbols, and standard library module names). Jones (2001) proposes a feature comparison category that compares two programs based on three *profiles*:

Physical profile characterises a program based on its physical attributes, such as the number of lines, words, and characters.

Halstead profile characterises a program based on its token types and frequencies. These includes the number of token occurrences (N), the number of unique tokens (n), and volume ($N \log_2 n$).

Composite profile a combination of the physical profile and the Halstead profile.

To detect plagiarism, the profiles of each program are calculated, and then normalised. The similarity of two programs is estimated by computing the Euclidean distance between their profiles (Jones 2001).

Prechelt et al. (2000) note that systems that use feature comparison may be very insensitive (can easily be misled), or very sensitive (producing many false

positives) since they ignore program structure. Offenders may easily add or remove comments, variables, or redundant lines of code to escape detection (Chen, Li, McKinnon & Seker 2002, Prechelt et al. 2000, Whale 1990).

2.2.2 Structure Comparison

This approach relies on the belief that the similarity of two programs can be estimated from the similarity of their structure. Programs are compared in two stages; the first stage parses the code and generates token sequences, while the second stage compares the tokens. Three systems that fall into this category are *Sim* (Gitchell & Tran 1999), *YAP3* (Wise 1996), and *JPlag* (Prechelt et al. 2000).

Sim

Sim detects similarities between programs by evaluating their correctness, style, and uniqueness (Gitchell & Tran 1999). Each program is first parsed using the *flex* lexical analyser, producing a sequence of integers (tokens). The tokens for keywords, special symbols, and comments are predetermined, while the tokens for identifiers are assigned dynamically and stored in a shared symbol table; whitespace is discarded. The token stream of the second program is grouped into sections, each representing a module of the program; each section is separately aligned with the token stream of the first program. An alignment of two strings is performed by inserting spaces between characters to equalise their length. An alignment scoring scheme is used to calculate similarity. This rewards matches involving two identifiers by two points, and other matches by one point. It also penalises mismatches involving two identifiers by two points, and other mismatches by one point. Gaps also attract a two-point penalty. *Sim* can handle name changes and reordering of statements and functions.

YAP3

YAP3 (Wise 1996) is another structure-based plagiarism detection system; it detects plagiarism through two phases. In the first phase, token sequences are generated from the source code. Comments and string constants are removed, and characters are converted to lower case. Functions are mapped to their base equivalents (such as `strncmp` to `strcmp`). In the second phase, the maximum, non-overlapping matches of the tokens sequences are then obtained using the running Karp-Rabin greedy string-tiling algorithm (Karp & Rabin 1987). *YAP3* is able to detect plagiarism with modified subsequences of lines and additional statements.

JPlag

JPlag (Prechelt et al. 2000) is a web-based detection system that uses a comparison algorithm similar to that of *YAP3*. In this system, the source code is parsed and converted into token strings. To minimise similarity by chance, *JPlag* includes some context of the program structure into the token strings, for example using the “`BEGIN_METHOD`” token to indicate an open brace at the beginning of a method and “`OPEN_BRACE`” to indicate other open braces. Whitespace, comments, and identifier names are ignored. The greedy string tiling algorithm is then used to compare token strings and identify the longest, non-overlapped common substrings. The result of the detection process is shown to the user in colour-coded HTML format.

Source Code

```
if (i==99) {
  int j=i+123;
}
```

Register Transfer Language (RTL)

```
...
(insn 14 12 15 (nil) (set (reg:CCZ 17 flags)
  (compare:CCZ (mem/f:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xfffffff]) [0 i+0 S4 A32])
    (const_int 99 [0x63])))) -1 (nil)
  (nil))
(jump_insn 15 14 16 (nil) (set (pc)
  (if_then_else (ne (reg:CCZ 17 flags)
    (const_int 0 [0x0]))
    (label_ref 22)
    (pc))) -1 (nil)
  (nil))
(note 16 15 17 0x4017d2c0 NOTE_INSN_BLOCK_BEG)
(note 17 16 19 NOTE_INSN_DELETED)
(insn 19 17 20 (nil) (parallel [
  (set (reg:SI 61)
    (plus:SI (mem/f:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffff]) [0 i+0 S4 A32])
      (const_int 123 [0x7b]))
    (clobber (reg:CC 17 flags))
  ]) -1 (nil)
  (nil))
(insn 20 19 21 (nil) (set (mem/f:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
  (const_int -8 [0xfffffff]) [0 j+0 S4 A32])
  (reg:SI 61)) -1 (nil)
  (expr_list:REG_EQUAL (plus:SI (mem/f:SI
    (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffff]) [0 i+0 S4 A32])
      (const_int 123 [0x7b]))
    (nil)))
  (nil)))
...
```

Optimised RTL

```
...
(insn 14 12 15 (nil) (set (reg:CCZ 17 flags)
  (compare:CCZ (reg/v:SI 61 [ i ])
    (const_int 99 [0x63])))) -1 (nil)
  (nil))
(jump_insn 15 14 16 (nil) (set (pc)
  (if_then_else (ne (reg:CCZ 17 flags)
    (const_int 0 [0x0]))
    (label_ref 21)
    (pc))) -1 (nil)
  (nil))
(note 16 15 17 0x4017d318 NOTE_INSN_BLOCK_BEG)
(note 17 16 19 NOTE_INSN_DELETED)
(insn 19 17 20 (nil) (parallel [
  (set (reg/v:SI 62 [ j ])
    (plus:SI (reg/v:SI 61 [ i ])
      (const_int 123 [0x7b]))
    (clobber (reg:CC 17 flags))
  ]) -1 (nil)
  (nil))
...
```

Figure 1: An example of source code (top left), the corresponding unoptimised RTL (bottom left), and the optimised RTL (right).

3 Plagiarism Detection Using Intermediate Code

Existing plagiarism detection systems, whether based on feature or structure comparison, are developed to detect plagiarism in a particular language such as C, Java, Pascal, or Scheme. We refer to this as *intra-lingual plagiarism*. However, none has been applied to detect *inter-lingual plagiarism*, where code in one language is plagiarised and rendered in another.

In this section, we describe our novel approach, that we call XPlag, to detect inter-lingual plagiarism by comparing the structure of intermediate code produced by a compiler suite.

A compiler typically processes source code in two passes (Hernandez-Campos 2002). In the front end pass, a source code file is scanned (lexical analysis), parsed (syntax analysis), and semantically analysed to produce intermediate code. In the back end pass, the intermediate code is optimised and its binary code (executable) is generated.

Since we wish to detect plagiarism that involves multiple languages, we need a compiler that supports more than one language. We refer to this type of compiler as a *compiler suite*.

3.1 The Compiler Suite

Two popular compiler suites are Microsoft Visual Studio .NET and the GNU Compiler Collection (GCC). Microsoft Visual Studio .NET is based on the .NET framework¹ and supports many languages, among them Microsoft Visual C#, Visual Basic .NET, Visual J#, and Visual C++ .NET. Program source code is compiled first by the appropriate front-end compiler to produce the the common intermediate language (CIL) — also known as the Microsoft Intermediate Language (MSIL). At run time, a Just-In-Time (JIT)

compiler is then used generate the executable (native) code from the intermediate code.

The popular GNU Compiler Collection (GCC)² also supports several languages including C, C++, Java, Fortran, and Objective C, and produces intermediate code in a common format (Jain, Sanyal & Khedker 2003). There is also ongoing work to integrate support for the .NET framework into GCC (Singer 2003). The GCC front end contains separate lexical analysis, syntax analysis, semantic analysis, and tree optimisation modules for each language; from this, a representation in a common intermediate code — the Register Transfer Language (RTL) — is generated. The back end optimises this RTL to produce machine code that is executable by the target machine.

The bulk of our work to date has focused on the GCC compiler suite using the C and Java languages, acknowledged to be the most popular³ of those supported by this compiler suite.

3.2 The Register Transfer Language

The GCC Register Transfer Language⁴ contains a series of instructions represented in nested parentheses. Each instruction contains a line number, a pointer to the previous instruction, and a pointer to the next instruction followed by expressions.

GCC provides three levels of compiler optimisation⁵. Figure 1 shows an example where the expressions in the optimised RTL — Figure 1 (right) — are simpler than the unoptimised RTL — Figure 1 (bottom left). In the optimised RTL, variable initialisation is performed by storing a value in a virtual register (represented by a **reg** token) rather than a virtual stack (represented by a **reg** and an offset value). This

¹<http://www.microsoft.com/net/basics/framework.asp>

²<http://gcc.gnu.org>

³<http://www.developer.com/lang/article.php/3390001>

⁴<http://gcc.gnu.org/onlinedocs/gcc-3.3.3/gccint/RTL.html>

⁵<http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

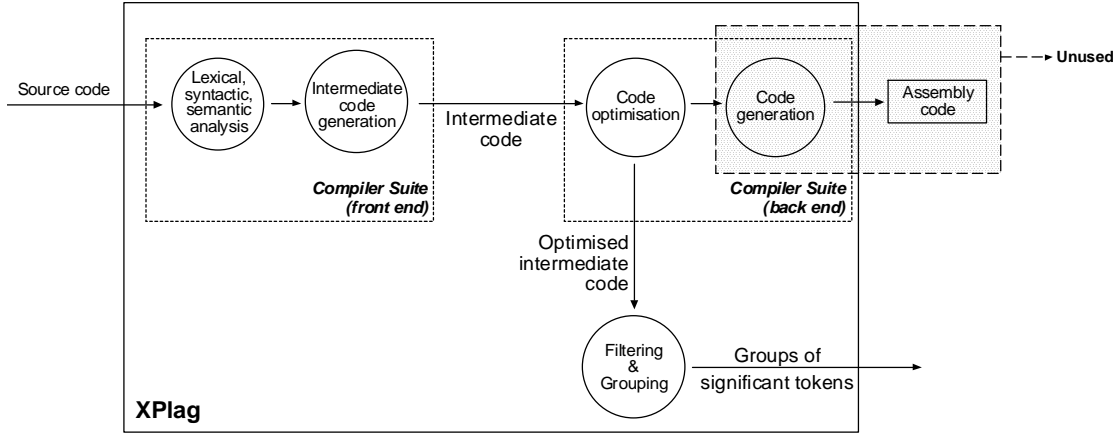


Figure 2: XPlag stages; the input source code is scanned and parsed by a compiler suite, producing the intermediate code. The intermediate code is optimised, filtered, and clustered, to produce a group of significant tokens.

| | | | | | |
|------------|-----------|------------------|---------|-------------|-----------------------------|
| insn | and | call | gt | truncate | NOTE_INSN_FUNCTION_BEG |
| const_int | parallel | call_insn | lt | ashiftrt | NOTE_INSN_FUNCTION_END |
| reg | clobber | call_placeholder | eq | ashift | NOTE_INSN_LOOP_BEG |
| set | plus | expr_list | le | lshiftrt | NOTE_INSN_LOOP_CONT |
| mem | minus | label_ref | ge | lshift | NOTE_INSN_LOOP_END_TOP_COND |
| code_label | unspec | symbol_ref | compare | sign_extend | NOTE_INSN_LOOP_END |
| use | jump_insn | pc | or | barrier | if_then_else |

Table 1: The list of RTL keywords we consider significant.

makes the RTL file — and consequently the search engine index — more compact. Selecting optimisation also causes function inlining, where the compiler integrates functions shorter than a threshold (the default is 600 lines) into the calling code.

In our preliminary research, we determined that the highest optimisation level brings the most benefit to our approach, since the RTL instructions are simplified. Moreover, optimisation allows straightforward detection of cases where function inlining is used to disguise copied code (Whale 1986). GCC also provides a compilation option to add debugging information in the intermediate code. Our observations indicate that this additional information is not helpful, and so we do not report experiments using this option.

3.3 The XPlag approach

The XPlag mechanism comprises two stages. In the indexing stage, all programs in the collection are converted into tokens, and token information is stored in an inverted index. In the detection stage, source code is used to query the index and produce a list of programs in the collection, ranked by decreasing similarity.

The internal process of XPlag is illustrated in Figure 2. The source code input is scanned and parsed by a compiler suite, producing the intermediate code. After optimisation, the intermediate code is filtered and clustered, producing groups of overlapping tokens — n -grams — as the output. We follow with a detailed description of this approach.

3.4 The Filtering Process

The RTL of a program contains a sequence of instructions, each containing a set of keywords. Some keywords, such as variable names, register names, and constants, are insignificant for two reasons. First, they do not represent the structure of a program; second, variable names and constants can be altered to disguise plagiarism. We therefore consider these keywords to be stop words, and filter them from the op-

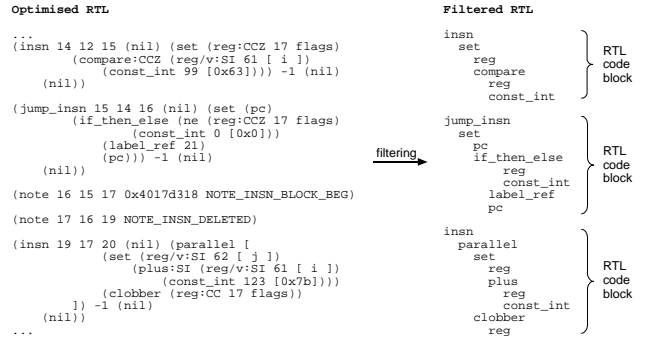


Figure 3: RTL example before filtering (left) and after filtering (right).

timised RTL⁶. The keywords we retain are listed in Table 1.

Figure 3 shows an example of RTL, before and after the filtering process. The indentation of the filtered RTL represents the depth of the nested expression in an RTL instruction. Constants, variable names, and machine modes — for example SI, indicating that the number is a single integer — are not retained. We discovered that filtered RTL for variable declarations, branching statements, and function calls have similar sequences of keywords across C and Java. However, we observed that the RTL structure of looping statements is different although the programs are similar. To address this, we keep the NOTE_INSN_LOOP_BEG and NOTE_INSN_LOOP_END keywords that are used to indicate the beginning and the end of a loop in the RTL generated from both languages.

3.5 The Mapping and Grouping Process

The significant keywords retained by the filtering process are between two and twenty-two characters, as listed in Table 1.

⁶We use the *flex* lexical analyser for this purpose.

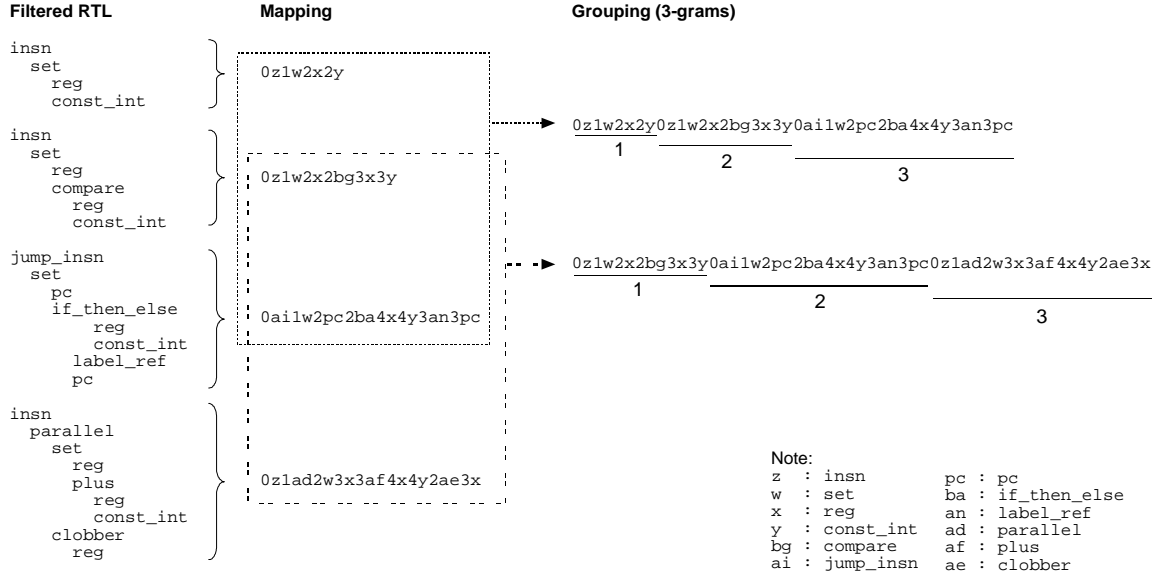


Figure 4: An example of the RTL mapping and grouping processes. Each significant keyword in the filtered RTL is converted to a one- or two-character code. Then, the mapped RTL is grouped into 3-grams. The number before each mapped RTL keyword represents the depth of indentation in the filtered RTL.

Two programs may contain similar instructions even where no plagiarism has occurred. Hence, the similarity of two programs should not be estimated by simply comparing the pairs of instructions contained in both programs. XPlag groups instructions into n -grams, where each gram contains n instructions, and each instruction contains a set of significant keywords.

We consider each RTL code block as a gram, with the indentation level indicated by a number. Figure 4 illustrates how the optimised RTL representation of a program is mapped and then grouped into 3-grams. The value of n is chosen empirically — we discuss this in further detail in Section 4.

3.6 The Search Engine

Plagiarism detection systems that are based on pairwise comparisons are not scalable; a good alternative approach is to index and query the tokens with a search engine (Burrows, Tahaghoghi & Zobel 2004). We incorporate a search engine into XPlag to perform two tasks:

1. In the indexing stage, the grouped RTL of the source code files in the collection is indexed.
2. In the detection stage, the grouped RTL of the query source code is used to search the index, and the programs of the collection are listed ranked by decreasing similarity to the query.

XPlag consists of two stages, namely the *indexing* and *detection* stages. In the indexing stage, a collection of programs is compiled with the compiler suite using the highest optimisation option, producing the intermediate code. This is then filtered, mapped, and grouped to n -grams, producing groups of significant keywords that are then indexed by the search engine. In the detection stage, the source code to be checked is similarly compiled, filtered, mapped, and grouped to n -grams, producing a group of significant intermediate code keywords. These keywords are then used as a query to the search engine, returning a list of similar programs ordered by decreasing similarity.

The similarity between a collection program and the query is estimated using a similarity measure or ranking function. The Okapi BM25 similarity function is highly effective for general text search, and is

defined as (Robertson & Walker 1999):

$$\sum_{t \in Q} w_t \cdot \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}} \cdot \frac{(k_3 + 1) f_{q,t}}{k_3 + f_{d,t}}$$

where:

$$w_t = \log_e \left(\frac{N - f_t + 0.5}{f_t + 0.5} \right), K = k_1 \cdot \left((1 - b) + \frac{b \cdot W_d}{W_D} \right)$$

| | | | |
|-------------|---------------------------|---------|----------------------|
| W_d = | document length | K_1 = | 1.2 |
| W_D = | average document length | k_3 = | infinite |
| N = | documents in collection | b = | 0.75 |
| $f_{q,t}$ = | query-term frequency | f_t = | collection frequency |
| $f_{d,t}$ = | within-document frequency | | |

BM25 is more suited to retrieval of text documents rather than to code, since the more often a query term occurs in a document, the higher the score given to the document. Chawla (2003) proposes the Plagi-Rank ranking function as more appropriate for code retrieval. This gives a higher score to documents in which query terms have the same frequency:

$$Score(Q, Q_d) = \sum_{t \in Q \cap Q_d} \left(\ln \frac{f_{qt}}{f_{dt}} + 1 \right) \cdot f_{qt} \cdot \frac{1}{W_d W_q} \text{ where } f_{qt} \leq f_{dt}$$

$$Score(Q, Q_d) = \sum_{t \in Q \cap Q_d} \left(\ln \frac{f_{dt}}{f_{qt}} + 1 \right) \cdot f_{qt} \cdot \frac{1}{W_d W_q} \text{ where } f_{qt} \geq f_{dt}$$

We evaluate our approach using both these measures and discuss the results in Section 4.

3.6.1 The Indexing and Detection Stages

In the indexing stage, the n -grams of the RTL of each program in the collection are collated into the TREC format⁷, and then indexed by the search engine.

In the detection stage, the RTL n -grams of the program to be checked are run as a query by the search engine, and a list of programs similar to the queried program is returned. We examine the top twenty documents.

The search engine provides an absolute similarity score for each document. This is unlikely to be meaningful to the average user, and so we instead refer to the Relative Percentage Similarity (RPS); this is the

⁷<http://www.seg.rmit.edu.au/zettair/zettair/doc/Readme.html>

ratio between the similarity score of a source code document and the similarity score of the query document to itself. To calculate this, the query source code must also be indexed by the search engine, and will be returned as the first answer. This represents the perfect match, with an RPS of 100%.

$$\text{Relative Percentage Similarity (RPS)}_i = \begin{cases} 100\% & \text{if } i = 1 \\ S_i/S_1 & \text{if } i > 1 \end{cases}$$

where i , RPS_i , and S_i represent the rank, Relative Percentage Similarity of the i -th document, and the search engine score of the i -th document, respectively. For example, if the search engine score of program file 0.c is 0.5 and the score of 67.c is 0.4328, then the relative percentage similarity of 0.c is 100% (because $i = 1$), and the score of program file 67.c is $0.4328/0.5=86.55\%$.

4 Experiments and Analysis

We continue with a discussion of our experiments, our analysis, and the external baseline we use to evaluate our technique.

All source code in our experiments was compiled using GCC version 3.3.3 on an Intel Pentium IV 2.4 GHz processor running the Linux SuSe 9.1 operating system. We also used version 0.6.1 of the Zettair⁸ search engine developed by the RMIT University Search Engine Group.

The aim of our experiments is twofold. First, to evaluate the performance of XPlag in detecting plagiarism among programs written in one particular language, that is, intra-lingual plagiarism. Second, to evaluate the performance of XPlag to detect inter-lingual plagiarism. For this purpose, we use three different collections of program source code:

1. **Collection-C**: contains 79 C programs from student submissions for an assignment in a course on Secure Electronic Commerce offered in Semester 2 2003.
2. **Collection-J**: contains 107 Java programs from the same assignment as **Collection-C**.
3. **Collection-X**: contains 206 programs from the combination of **Collection-C** and **Collection-J** and with the addition of ten equivalent pairs of C and Java program files from the Web. This collection was used to see how well XPlag can detect plagiarism across C and Java programs.

Some programs in **Collection-X** were obtained by translating programs written in the C language to the Java language using the Jazillian online translation tool⁹ to reflect an approach students may take when copying. We could not find any translation tools to translate programs written in the Java language to the C language. We found that some minor editing is still required to allow the code translated by Jazillian to be compiled under GCC. For example, we have to add the ‘static’ keyword before each function called from the static main() function, and replace ‘int int’ to ‘int’ because Jazillian translates ‘unsigned int’ as ‘int int’.

There are other automatic C-to-Java translation programs available¹⁰, but most, such as C2J and Ephedra produce output that is clearly machine-generated code.

⁸<http://www.seg.rmit.edu.au/zettair/>

⁹<http://www.jazillian.com>

¹⁰<http://www.jazillian.com/competition.html>

4.1 Ground Truth and Evaluation

To evaluate the effectiveness of our approach, we need the ground truth, that is, a list of programs that are known to be plagiarised. The ground truth of each set was determined as follows:

1. For **Collection-C**, we used exhaustive manual comparisons. There were twelve groups of programs that we regarded as copied.
2. For **Collection-J**, because of time constraints, we manually verified pairs of similar programs identified by JPlag. We found seven groups of plagiarised programs.
3. For **Collection-X**, we used the known cross-plagiarised programs downloaded from the Web, and the pairs which we translated using Jazillian.

The difference in the way the ground truth for each collection was prepared is likely to affect the absolute performance of JPlag and XPlag. Nevertheless, we believe that the experimental results reflect the relative performance of the two approaches within each collection.

To quantify the performance of our detection, we use the standard information retrieval measures of *precision* and *recall* (Witten, Moffat & Bell 1999). *Precision* is the ratio of documents retrieved that are relevant, while *recall* is the proportion of the relevant documents that have been retrieved.

$$\text{Precision (P)} = \frac{\text{relevant documents retrieved}}{\text{retrieved documents}}$$

$$\text{Recall (R)} = \frac{\text{relevant documents retrieved}}{\text{relevant documents}}$$

In the context of source code plagiarism detection, precision represents the number of plagiarised programs at some point in the returned list. The higher the precision, the more accurate the detection (fewer false positives). Recall represents the number of plagiarised programs detected out of all plagiarised programs in the collection. The higher the recall, the fewer copied programs escape detection (fewer false negatives).

Three measures derived from precision and recall include *R-precision*, *Precision@n*, and *interpolated precision-recall*. *R-precision* is the precision at the R -th program on the list, where R is the number of correct answers for the query; *Precision@n* is the precision at the n -th program on the list; finally, the interpolated precision at a particular recall level is the highest precision observed at that or any higher recall level. Interpolated precision-recall is usually shown at the eleven 10% steps of recall from 0% to 100%.

Important to us are the Precision@2 (P@2), Precision@5 (P@5), Precision@10 (P@10), R-Precision (R-P), and interpolated precision-recall scores. Precision@2 is useful for measuring how effective XPlag ranks a copied program at the second position on the list. Since the first program on the list is always the query program, Precision@2 is effectively the precision when only the most similar collection document to the query is examined. Precision@5 is used to evaluate the accuracy of XPlag in returning the top five programs; this is a useful cut-off point, assuming that a program is unlikely to have more than four other derivatives in the collection.

We also evaluate the precision and recall values at every 5% of the Relative Percentage Similarity (RPS) to estimate an RPS value that can be used by users as a good cut-off value to stop manual verification,

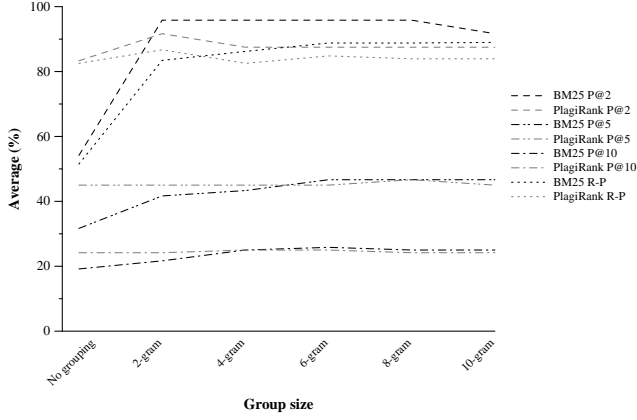


Figure 5: Performance comparison of the BM25 and PlagiRank similarity measures for varying n -gram sizes on Collection-C.

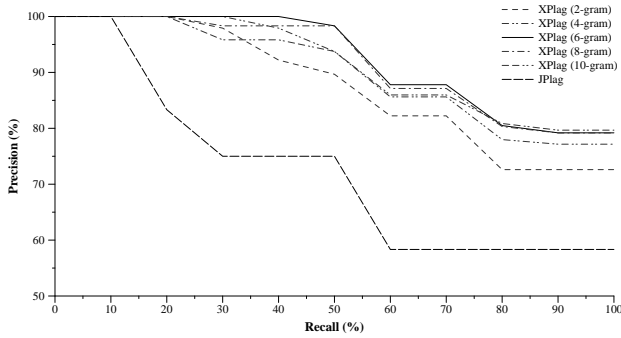


Figure 6: Interpolated precision-recall using the BM25 similarity measure and various grouping sizes on Collection C.

although this is likely to be somewhat dependent on the preference of the user for high precision or for high recall.

To test whether differences in performance are significant, we use the Wilcoxon signed rank test at the 95% confidence level.

4.2 Experiments with Collection-C

In our first series of experiments, we aim to evaluate the performance of XPlag in detecting plagiarism involving only C programs (Collection-C), identify the best grouping size to be used, and investigate which ranking function, BM25 or PlagiRank, produces better results. Our query set contains twelve programs taken from our ground truth. We find that grouping keywords into n -grams improves the precision of XPlag, as shown in Figure 5. Although we initially expected the PlagiRank similarity measure, specifically designed for plagiarism detection, to outperform BM25, the reverse is true for most combinations of n -gram sizes and evaluation measurements. Interestingly, all XPlag results are better than the JPlag baseline.

In our experiment, the BM25 with a group size of 6 produces the highest precision of all evaluation measurements. Figure 6 shows the interpolated precision at standard recall levels using the BM25 and 6-grams. We see that precision drops sharply after 50% recall (when half the incidents of known plagiarism instances have been retrieved), but remains above 75% for most n -gram sizes tested.

For comparison, we performed plagiarism detection using JPlag with the sensitivity value — or the minimum match length — set to 6; this value is equivalent to the grouping size of 6 that we used for XPlag.

| Average Precision at | XPlag (%) | JPlag (%) |
|----------------------|-----------|-----------|
| 2 | 100.00 | 79.00 |
| 5 | 44.00 | 32.00 |
| 10 | 22.00 | 16.00 |
| R | 80.00 | 71.00 |

Table 2: Performance comparison of XPlag and JPlag on Collection-C.

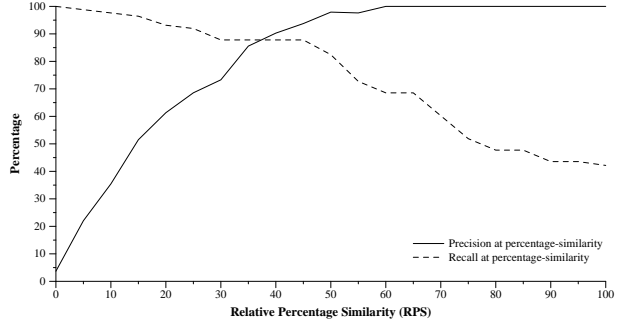


Figure 7: Average precision and recall values at 5% intervals of Relative Percentage Similarity using the BM25 and 6-grams on Collection-C.

Table 2 shows that XPlag outperforms JPlag in all evaluation measurements. Furthermore, the interpolated precision of XPlag at standard recall levels is significantly higher than JPlag, as shown in Figure 6. Table 2 also shows that XPlag accurately returns copied C programs at the second position on the list (Precision@2 is 100%).

In application, manual examination of highly ranked documents is likely to be impractical, and so we explore how the Relative Percentage Similarity value relates to the tradeoff between recall and precision. We plot the average precision and recall values at 5% Relative Percentage Similarity intervals in Figure 7. At 0% RPS, the recall reaches 100% (because all programs in the collection are listed) and the average precision is around 4% (because many false positives are returned). When RPS is equal to or greater than 60%, precision reaches 100% (no false positives), but recall decreases from 66% (at 60% RPS) to 48% (at 100% RPS). Requiring matches to have a higher RPS results in more false negatives.

4.3 Experiments with Collection-J

In our next series of experiments, we evaluated the performance of XPlag in detecting plagiarism involving only Java programs (Collection-J). We inspected the JPlag detection result to obtain groups of copied programs, and verified seven program pairs as copied.

Using the copied pairs as the query set, we find the result to be consistent with our experiments on Collection-C. Figure 8 demonstrates that grouping improves the performance of XPlag, and that BM25 produces better results than PlagiRank for most combinations of grouping sizes and evaluation measurements. The figure also shows that BM25 with group sizes of 4 and 6 produces the highest precision for all evaluation measures, although Figure 9 shows that using 4-grams leads to the highest interpolated precision results.

Table 3 compares the performance of XPlag and JPlag for detection of plagiarism among Java programs. Since the ground truth of Collection-J was generated from the JPlag detection result, JPlag performance represents ideal performance here; this is shown as a constant 100% interpolated precision-recall in Figure 9. While XPlag appears to perform

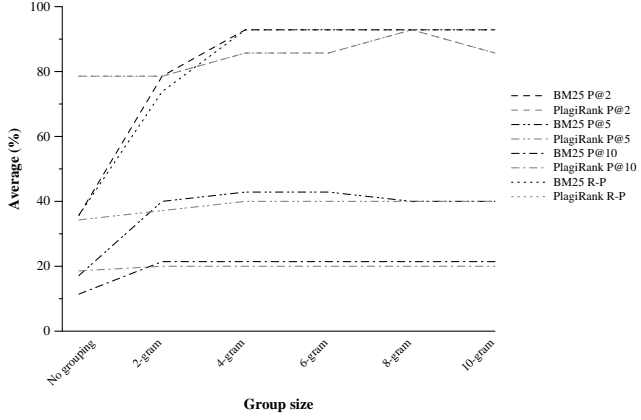


Figure 8: Performance comparison of the BM25 and PlagiRank similarity measures for varying n -gram sizes on Collection-J.

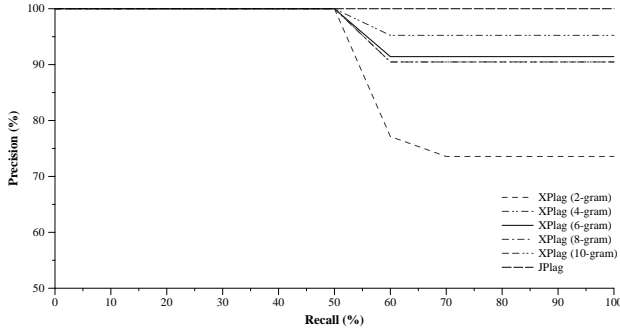


Figure 9: Interpolated precision-recall using the BM25 similarity measure and various grouping sizes on Collection J.

less than this ideal, the difference is statistically insignificant¹¹. XPlag successfully returns all occurrences of plagiarism in the collection within the first five answers — the Precision@5 is equal for both XPlag and JPlag — and returns copied Java programs at the second position on the list with an average precision of 92.86%.

Figure 10 shows that the precision increases at a constant rate from 10% RPS, and reaches 100% at 55% RPS. From 25% RPS onwards, recall decreases gradually from 100% to 48% at 100% RPS. We observe that for both **Collection-C** and **Collection-J**, precision is greater than 90% (less than 10% false positives) and recall is greater than 80% (less than 20% false negatives) at 50% RPS.

4.4 Experiments with Collection-X

We have shown that XPlag can detect intra-lingual plagiarism with reasonable precision and recall values. To investigate the XPlag performance in detecting inter-lingual plagiarism, we use **Collection-X** and two different query sets: **Queryset-C** (containing only the ten C programs); and **Queryset-Java** (containing only the ten Java programs).

To explore whether we can use JPlag again as our baseline, we tried to perform plagiarism detection on **Collection-X** using the JPlag C/C++ and Java parsers in turn. However, neither was able to reveal inter-lingual plagiarism. The detection using the Java parser excludes 97 programs due to unsuccessful compilation; this is understandable since the Java parser cannot process the submitted C programs. In

¹¹We used the Wilcoxon signed rank test at the 95% confidence level to compare the interpolated precision at standard recall levels of XPlag (4-grams) and JPlag.

| Average Precision at | XPlag (%) | JPlag (%) |
|----------------------|-----------|-----------|
| 2 | 92.86 | 100.00 |
| 5 | 42.86 | 42.86 |
| 10 | 21.43 | 21.43 |
| R | 92.86 | 100.00 |

Table 3: Performance comparison of XPlag and JPlag on Collection-J.

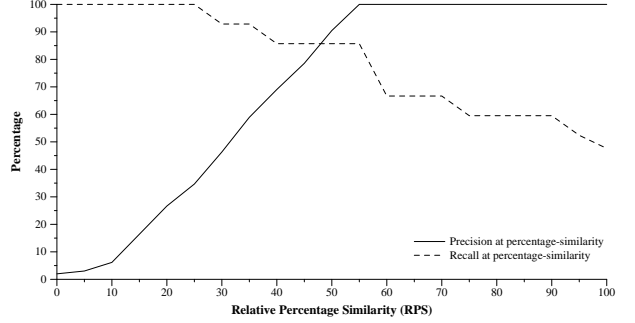


Figure 10: Average precision and recall values at 5% interval of Relative Percentage Similarity using the BM25 and 4-grams on Collection-J.

contrast, although the C parser can process most of the Java programs, only one incidence of inter-lingual plagiarism is reported, and that because one of the subroutines in both programs is identical. While this is not surprising — after all, JPlag is not designed to detect inter-lingual plagiarism — it leaves us with no external baseline for **Collection-X**.

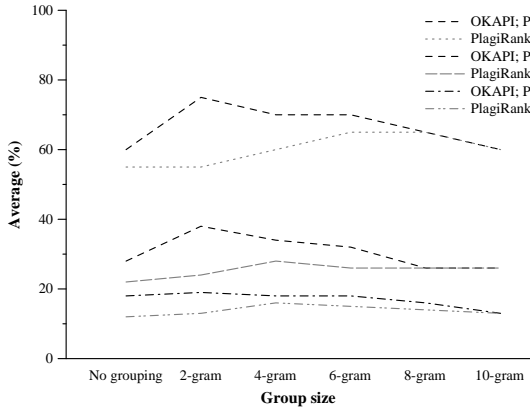
Figure 11 and Figure 12 show that BM25 with a group size of 2 produces the best Precision@2 and interpolated precision-recall for both query sets. Using **Queryset-C**, precision increases sharply from 0% to 30% RPS, as shown in Figure 13 (a); while using **Queryset-Java**, precision increases sharply from 0% to 40% RPS, as shown in Figure 13 (b). Precision reaches 100% for both query sets when RPS is equal to or greater than 60%, while recall is above 90% when RPS is equal to or less than 20%.

5 Discussion and Future Work

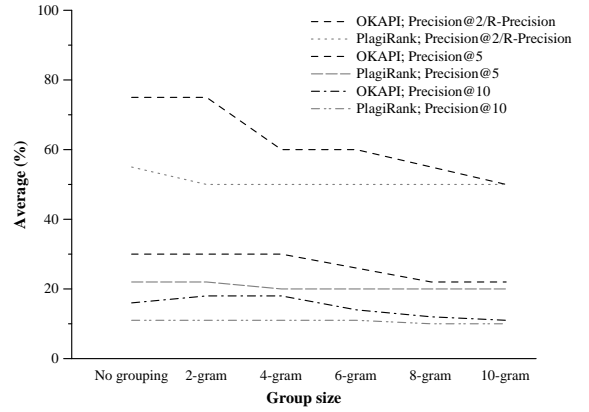
Plagiarism is a serious and widespread problem. Several approaches have been proposed to reveal plagiarism in source code, but these only aim to detect plagiarism involving one programming language. In this paper, we have described our novel approach, XPlag, to detect inter-lingual plagiarism by inspecting the intermediate code produced by a compiler suite. Using three different collections and employing the popular JPlag system as our baseline, we have shown that XPlag can detect intra-lingual plagiarism in and Java programs with reasonably good precision. Significantly, we have also shown that XPlag can detect inter-lingual plagiarism, albeit with lower accuracy than for intra-lingual plagiarism.

While the RTL for variable declarations, function calls, and branching statements of C and Java programs are similar, the RTL of Java programs often contains instructions for processing classes and function calls of the standard Java library. For example, there are fewer RTL instructions for the C ‘`printf()`’ function call than the Java ‘`System.out.println()`’ method call. We believe that the performance of XPlag can be improved by enhancing the filtering process to remove insignificant instruction groups, and by classifying equivalent function calls in C and Java RTL.

There are some limitations to our approach that

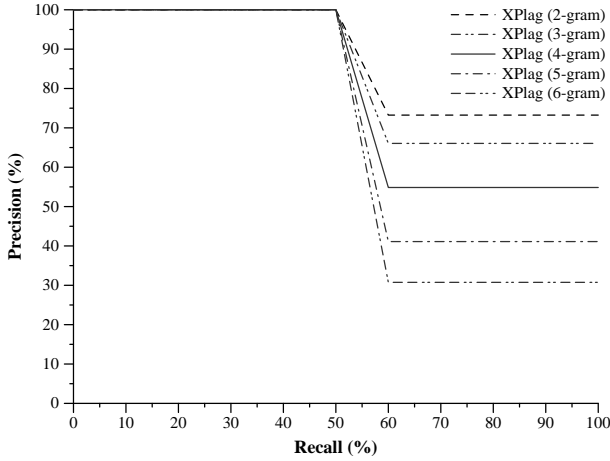


(a)

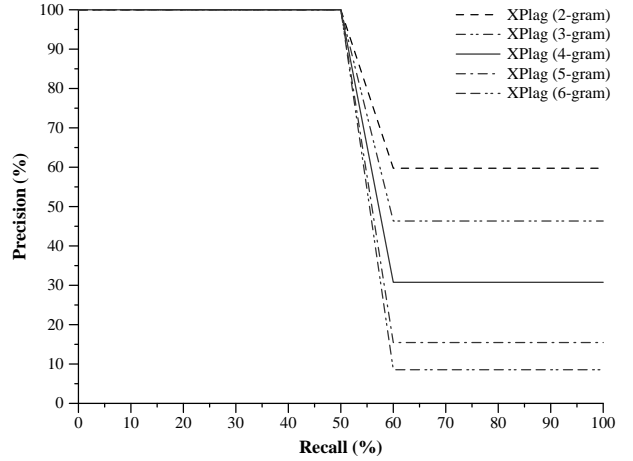


(b)

Figure 11: Performance comparison of the BM25 and PlagiRank similarity measures for varying n -gram sizes on Collection-X using (a) Queryset-C and (b) Queryset-Java.

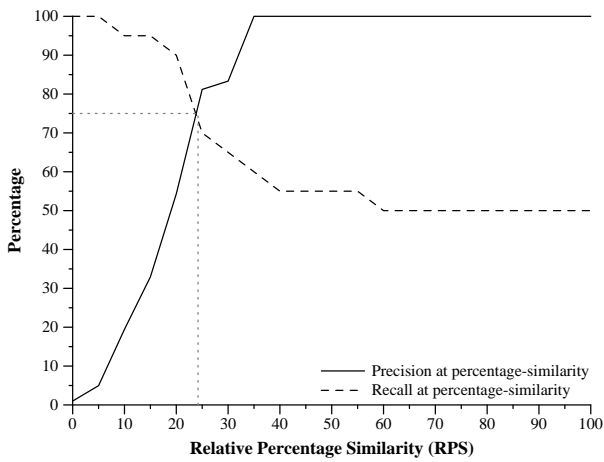


(a)

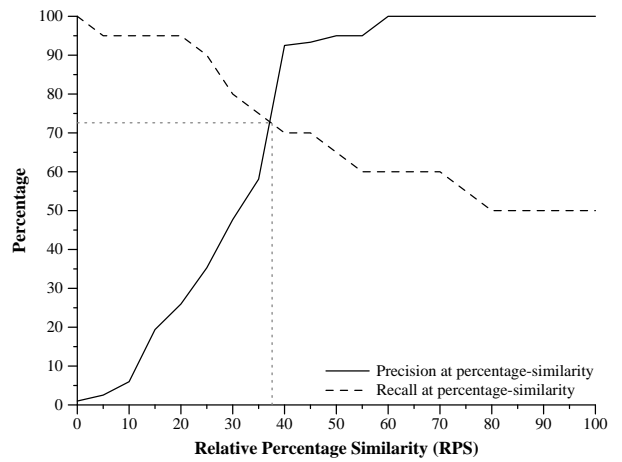


(b)

Figure 12: Interpolated precision-recall using the BM25 similarity measure and various grouping sizes on Collection X using (a) Queryset-C and (b) Queryset-Java.



(a)



(b)

Figure 13: Average precision and recall values at 5% interval of Relative Percentage Similarity using the BM25 and 2-grams on Collection-X using (a) Queryset-C and (b) Queryset-Java.

should be addressed in future work. First, all submitted programs must be successfully compiled by the compiler suite; if a program cannot be successfully compiled, it must be corrected manually for further processing. In an educational setting, this might perhaps be addressed by penalising non-compilable code as a matter of assessment policy.

Second, the results of our experiments are only valid when using the GCC compiler suite for plagiarism detection involving programs written in the C and Java languages. Preliminary experiments indicate that the intermediate language produced by the Microsoft Visual Studio .NET compiler suite can be used for intra-lingual plagiarism detection. We plan detailed experiments with this compiler suite.

Third, while the collections we used are realistic for a typical computing courses, we must investigate how the effectiveness of our approach behaves across other collections, and also for very large repositories of historical or crawled program source code. Associated work on intra-lingual plagiarism detection indicates that the underlying approach scales well in both effectiveness and efficiency (Burrows et al. 2004, Chawla 2003).

Finally, we plan to explore the alternative approach of using tokens produced by existing approaches (for example *Sim* or *JPlag*) instead of using intermediate code.

Overall, we believe that our approach can greatly help address the problem of inter-lingual plagiarism, and in this way help reduce the incidence of code plagiarism in general.

Acknowledgments

We thank Professor Justin Zobel and Guido Mapohl for their advice on this project, and Steven Burrows for donating Collection-C.

References

- Burrows, S., Tahaghoghi, S. M. M. & Zobel, J. (2004), Efficient and effective plagiarism detection for large code repositories, in 'G. Abraham and B.I.P. Rubinstein Editors, Proceedings of the Second Australian Undergraduate Students' Computing Conference (AUSCC04)', pp. 8–15.
- Chawla, M. (2003), An indexing technique for efficiently detecting plagiarism in large volumes of source code, Honours thesis, RMIT University, Melbourne, Australia, October.
- Chen, X., Li, M., McKinnon, B. & Seker, A. (2002), 'A theory of uncheatable program plagiarism detection and its practical implementation'. URL: <http://www.cs.ucsb.edu/~mli/sid.ps> [13 August 2005].
- Gitchell, D. & Tran, N. (1999), *Sim*: a utility for detecting similarity in computer programs, in 'Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education', ACM Press, pp. 266–270.
- Hernandez-Campos, F. (2002), 'Lecture 31: Building a runnable program'. URL: <http://www.cs.unc.edu/~stotts/COMP144/lectures/lect31.pdf> [13 August 2005].
- Hoad, T. & Zobel, J. (2003), 'Methods for identifying versioned and plagiarised documents', *Journal of the American Society of Information Science and Technology* **54**(3), 203–215.
- Jain, N., Sanyal, A. & Khedker, U. (2003), Retargeting GCC for cradle's DSE processor, Technical report, Department of Computer Science & Engineering, Indian Institute of Technology, Bombay, Bombay, India.
- Jones, E. L. (2001), Metrics based plagiarism monitoring, in 'Proceedings of the Sixth Annual CCSC Northeastern Conference, Middlebury, Vermont', pp. 1–8.
- Karp, R. M. & Rabin, M. O. (1987), 'Efficient randomized pattern-matching algorithms', *IBM Journal of Research and Development* **31**(2), 249–260.
- Prechelt, L., Malpohl, G. & Philippsen, M. (2000), *JPlag*: Finding plagiarisms among a set of programs, Technical Report 2000-1, Fakultät für Informatik Universität Karlsruhe, D76128 Karlsruhe, Germany.
- Robertson, S. E. & Walker, S. (1999), Okapi/Keenbow at TREC-8, in 'The Eighth Text Retrieval Conference (TREC-8)', pp. 151–162.
- Sheard, J., Dick, M., Markham, S., Macdonald, I. & Walsh, M. (2002), Cheating and plagiarism: Perceptions and practices of first year IT students, in 'Proceedings of the Seventh Annual Conference on Innovation and Technology in Computer Science Education', pp. 183–187.
- Singer, J. (2003), GCC .NET—a feasibility study, in 'Proceedings of the First International Workshop on C# and .NET Technologies', University of West Bohemia, Plzen, Czech Republic.
- Whale, G. (1986), Detection of plagiarism in student programs, in 'Proceedings of the Ninth Australian Computer Science Conference, Canberra', pp. 231–241.
- Whale, G. (1990), 'Identification of program similarity in large populations', *The Computer Journal* **33**, 2.
- Wise, M. J. (1996), 'YAP3: Improved detection of similarities in computer program and other texts', *SIGCSE Bulletin* **28**(1), 130–134.
- Witten, I. H., Moffat, A. & Bell, T. C. (1999), *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers, second edition.
- Zobel, J. (2004), "Uni cheats racket": a case study in plagiarism investigation, in 'Proceedings of the Sixth Conference on Australian Computing Education', Australian Computer Society, Inc., pp. 357–365.
- Zobel, J. & Hamilton, M. (2002), 'Managing student plagiarism in large academic departments', *Australian Universities Review* **45**(1), 23–30.