Comparing C Code Trees

Warren Toomey

Faculty of IT, Bond University

<wkt@staff.bond.edu.au>

ABSTRACT

Ctcompare is a tool to lexically compare two C code trees for possible code copying. This paper describes the motivation behind the creation of the tool, discusses some of the issues that must be faced when attempting to compare code trees, and highlights some of the design aspects of the tool.

1. Introduction

There are several reasons for constructing a tool which can automate the comparison of program source trees. For example, you may be a large software house which has accumulated several packages through mergers; if a competitor sues you for using their copyright code, the task of manually determining if this is a valid claim would be arduous. In an academic setting, a comparison tool would help to catch students who have plagiarised from other students or who have colluded on an assignment. And in the newly-developing field of computing history, a comparison tool could help to explicate the development of a computing artifact.

The motivation for the construction of *Ctcompare*, the code comparison tool described here, is more prosaic. In March 2003, the SCO Group sued IBM for “misusing and misappropriating SCO’s proprietary software; inducing, encouraging, and enabling others to misuse and misappropriate SCO’s proprietary software; and incorporating (and inducing, encouraging, and enabling others to incorporate) SCO’s proprietary software into open source software offerings.”[10] While the original complaint did not explicitly mention copyright violation as the means of software misappropriation, follow-up press releases from SCO and interviews with prominent SCO executives did:

> *Although SCO’s claims about Linux developers copying from SCO’s proprietary UnixWare have been vague in the past, this time [Chris] Sontag specifically claimed that there is “significant copyrighted and trade secret code within Linux”. When asked for examples of infringement, Sontag said, “It’s all over the place” but did not characterize any one subsystem as containing more infringing code than others. Infringement is present not only in distributions and vendor kernels, but in the official kernel available from kernel.org. Code has been “munged around solely for the purpose of hiding the authorship or origin of the code”, he said. … “We specifically excluded the BSD-derived code”, Sontag said. “There is post-BSD UnixWare source code origined [sic] with SCO, and that is of issue.”[5]*

> *Replacing the illegal code seems unimaginable, even if we would be the first to approve such a solution. But we’re talking about millions of lines of code and not a few dozen. On top of that, the pieces that were taken are precisely what makes Linux a viable solution for enterprise deployment, like SMP and NUMA.[1]*

`\end{enumerate}`

At a trade show in August 2003, SCO showed two snippets of source code which they claimed were illegally copied from UNIX[1] into the Linux kernel. Subsequent analysis showed that one snippet of code was indeed UNIX code which had been released under a BSD license by Caldera, and the second snippet of code originated in BSD and was actually not UNIX code[7, 9, 4].

2. Proving or Disproving Code Copying

Program source code can be copied in at least two ways[2]:

- Literal copying: the direct copying of lines of source from one system to another, with or without comments, and with possible rearranging by hand or with tools such as indent(1).
- Non-literal copying: the transfer of methods, structures and sequences from one system to another. This includes data structures, program interfaces and algorithms.

1. UNIX is a trademark of the X/Open Group.
2. Issues such as trade secret and patent violations are not covered in this paper.

While the first form of copying is easy to spot, the latter often requires human judgement. Many of the federal courts in the United States have adopted the following test to determine if source code has been improperly copied:

> *In 1992, the Second Circuit Federal Court of Appeals … developed a three-part test for determining whether software is infringed under the copyright laws. The test, which has become known as the "abstraction/filtration/comparison" test, is based upon a similar copyright infringement test enunciated by Judge Learned Hand in 1930 in Nichols v. Universal Pictures Corp.*
>
> *In the first step of the revised … test, the computer program is divided into its various levels of abstraction.*
>
> *The second test, the "filtration" step, entails examining the structural components of the software at each level of abstraction to determine (1) whether their particular inclusion at that level was {}``idea'' or was dictated by considerations of efficiency, (2) whether their inclusion was required by factors external to the program itself, such as a required data input or output protocol, or (3) whether the structural components were taken from the public domain. If a particular structural component at each level of abstraction satisfies any of the three criteria, then it is non-protectable expression, and is not considered in the final step of the test, described next.*
>
> *The third and final step, the "comparison" step, involves comparing the expression left after the filtration step at each level of abstraction to the accused software in order to determine whether there is substantial similarity between the two. If there is substantial similarity, and it can be shown that the developer of the accused software had access to the original software, then copyright infringement may be found.[3]*

While this approach provides a rigorous test once suspect code has been found, it does not find potentially misappropriated source code in the first place.

3. Automating Code Comparison

The analyses of the purported code copying from UNIX into Linux described in Section 1 were done by hand. In many cases, analysis was done by someone who had intimate knowledge of UNIX kernel source. However, performing a similar analysis to prove if and where UNIX source had been improperly copied into Linux would be an impossible task: both systems are comprised of many millions of lines of code. Moreover, SCO has indicated their belief that code had been "munged around solely for the purpose of hiding the authorship or origin of the code". This would complicate the task still further.

Another difficulty to be faced is the confidentiality of proprietary source code. In the SCO vs. IBM case, the Linux source code is freely available but the UNIX source code is not. This appears to limit the comparison of source code to those who have legal access to the UNIX source code. For the same reason, if source code for an Open Source program was misappropriated into a proprietary application, the authors of the original program may not be able to offer proof that the code had been copied.

In September 2003, Eric S. Raymond showed that there was a way around the proprietary software impasse with a C code comparison tool called *comparator*:

> *Comparator is a program for quickly finding common sections in two or more source-code trees. … [It] works by first chopping the specified trees into overlapping shreds (by default 3 lines long) and computing a hash of each shred. The resulting list of shreds is examined and all unique hashes are thrown out. Then comparator generates a report listing all cliques of lines with duplicate hashes, with overlapping ranges merged. … A consequence of the method is that comparator will find common code segments wherever they are. It is insensitive to rearrangements of the source trees.[8]*

Hashing the two sets of source code before comparison allows the hashes for a proprietary system to be openly published without the disclosure of the actual source code. A person with legal access to UNIX System V or UnixWare source code could publish their hashes and allow SCO's claims of inappropriate code copying to be verified.

4. Drawbacks of the Comparator Approach

Comparator suffers from several drawbacks which limit it to the detection of simple line-by-line code copying. It cannot deal with source lines which have been split or concatenated. Comparator also cannot detect code copying where variables or functions have been renamed. Examples of these are shown below:

Original Code

```
for (i = 0; error == -1 && execsw[i]; ++i) {
  if (execsw[i]->ex_imgact == NULL ||
    execsw[i]->ex_imgact == img_first) {
      continue;
  }
  error = (*execsw[i]->ex_imgact)(imgp);
}
if (error) {
  if (error == -1)
    error = ENOEXEC;
  goto exec_fail_dealloc;
}
```

Copied Code

```
for (c = 0; err == -1 && exsw[c]; ++i) {
  if (exsw[c]->ex_imgact == NULL ||
    exsw[c]->ex_imgact == img_first) {
      continue;
  }
  err = (*exsw[c]->ex_imgact)(imgp);
}
if (err) {
  if (err == -1) err = ENOEXEC;
  goto exec__dealloc;
}
```

Simple code changes dealing with whitespace, case sensitivity and removal/ inclusion of comments can be handled with heuristics, but the line-based approach of Comparator brings serious comparison limitations in the face of SCO's assertion of code "munging'".

5. A Lexical Approach to Code Comparison

Eric Raymond's Comparator tool (with its use of hashes) inspired me to design my own C code comparison tool. Dissatisfied with the line-based approach, I wanted to construct a tool which would permit non-UNIX source code holders to verify or disprove SCO's assertion of code copying from UNIX into Linux. To be useful, the comparison tool would need to meet these requirements:

- The tool must detect the rearrangement of source code, even if the rearrangement is not line-by-line.
- The tool must provide an exportable code representation which does not disclose the original source code, so as to allow others to verify any code comparison.
- The tool must be reasonably fast: O(n^2) or better where n is the number of lines of source to be compared.
- The tool should be able to detect the renaming of variables and functions to some extent.
- If possible, the tool should detect some non-literal code copying, such as the use of basic algorithmic elements to perform a task (loops, if/else or case statements).

Requirement #1 rules out a line-based approach. Instead, for my *Ctcompare* tool I chose a lexical approach to code comparison where each C source file is broken into a list of tokens or *lexemes*[3], and then the two lists of lexemes are compared for any similarity.

5.1 Compiling the Code Does Not Work

C source code is normally converted by a compiler into a "parse tree" representation for semantic analysis before it is then converted into an equivalent low-level code. One might consider the comparison of two parse trees to be a suitable approach, but in practice this tends to fail for many reasons.

C code is usually processed by the C pre-processor before being passed to the compiler. Different systems will have distinct sets of header files. Macro and structure definitions will differ between systems, thus rendering the parse trees for even the same program different. Programs themselves often use pre-processor directives to omit sections of code at compile time (`#ifdef` … `#endif`), thus removing some source code from comparison analysis. Finally, any bugs or misfeatures in the source code may prevent a full parse tree from being constructed, again removing some source code from comparison analysis.

A full semantic parse tree, therefore, is not suitable for code comparison. This is why I chose to break each C source file into a list of lexemes. At the same time, the source files would not be passed through the C pre-processor, to prevent any loss of source code.

5.2 Modifying the Lexical Analyser

By taking a lexical approach and excluding the pre-processor, existing C lexical analysers could not be used as they all assume that pre-processor directives will have been removed. To overcome this, I chose an Open Source C lexical analyser, *cslang.l* from the CSlang program by Tudor Hulubei, and modified it to deal with the following pre-processor directives:

```
#[ \t]*define
#[ \t]*elif
#[ \t]*else
#[ \t]*endif
#[ \t]*error
#[ \t]*ifdef
#[ \t]*if
#[ \t]*ifndef
#[ \t]*include
#[ \t]*line
#[ \t]*pragma
```

3. Lexeme: A minimal lexical unit of a language. Lexical analysis converts strings in a language into a list of lexemes.[12]

```
#[ \t]*undef
#[ \t]*warning
```

6. What Tokens to Export?

The modified C lexical analyser knows about 100 separate C language tokens. The first version of Ctcompare simply tokenised each input file and exported a token stream, with a byte representing a single token. Comments were excluded from the output, and line breaks were kept but not used in code comparison. For example, the C code fragment

```
void print_word(char *str)
{
  char *c;
  c=str;
  while ((c!='\0') && (c!=' ')) putchar(c++);
  putchar('\n');
}
```

would be represented by the token stream

VOID IDENTIFIER OPENPAREN CHAR MULT IDENTIFIER CLOSEPAREN LINE

OPENCURLY LINE

CHAR MULT IDENTIFIER SEMICOLON LINE

IDENTIFIER EQUALS IDENTIFIER SEMICOLON LINE

WHILE OPENPAREN OPENPAREN IDENTIFIER NOT EQUALS CHARCONST …

However, this loses too much semantic information, allowing the code fragment to be seen as "identical" to

```
void do_junk(char *g)
{
  char *c;
  c=g;
  while ((c!='q') && (c!='f')) atoi(g++);
  sqrt('?');
}
```

Some information about the identifiers in a C file must be exported, but in such a way that the full source code is not disclosed. A solution like the following would not be satisfactory to export proprietary source code:

VOID IDENTIFIER print_word OPENPAREN CHAR MULT

IDENTIFIER str CLOSEPAREN LINE

OPENCURLY LINE

CHAR MULT IDENTIFIER c SEMICOLON LINE

IDENTIFIER c EQUALS IDENTIFIER str SEMICOLON LINE

WHILE OPENPAREN OPENPAREN IDENTIFIER c NOT EQUALS CHARCONST '\' …

To avoid this, I then chose to enumerate each identifier as they appear in the file. In the first code fragment above, `print_word` is identifier #1, `str` is #2, `c` is #3 and `putchar` is #4. This keeps some contextual information about identifiers in the file without revealing their names.

This has been since improved upon in the current version of Ctcompare. 16-bit hashes of each identifier and constant are now stored in the token stream. The hashes obscure the actual identifier names, but permit the detection of identifier name reuse in two different code trees.

7. 1st Implementation: Proof of Concept

With the lexical analysis and token stream exporting done, it was time to move to the actual comparison of token streams. The first code comparison implementation was proof of concept:

- Take two tokenised streams, and treat them as "strings".
- For each string in the first stream, find all matching strings starting with the same token in the second stream.

This brute-force implementation is O($n \times m$), where n & m are the string lengths. It was also slowed down by dealing with the "LINE" tokens embedded in the input streams, and by poor loop design. The implementation unfortunately would miss some matches due to false skipping. Consider the following two strings:

HELLOTHEREHOWAREYOU?
WHATCELLOBEWARELOTHERE?

Here (H)ELLO matches (C)ELLO, but the program should not then skip forward to THERE, as (L)LOTHERE matches (E)LOTHERE. There were many false matches due to the initial choice to omit identifier values, and due to common C features such as:

```
#include < word . word >
#include < word . word >
```

and

```
int id [ ] = ( num, num, num,
     num, num, num, num, ...
```

The next version was modified to include the enumeration of identifier values and a run-time switch to ignore C pre-processor directives in the input streams. The bottom 16-bits of numeric constants were encoded into the token stream to reject non-matching numeric constants. However, even with these changes there were still too many false matches on common C constructs such as

```
for (d=0; d < NDRV; d++)
```

and

```
for (i=0; i< j; i++)
```

where `NDRV` is a constant defined at the top of the file or elsewhere, and `j` is a local variable.

8. Code Isomorphism

By recording contextual information about identifiers and constants (by enumerating them in order of appearance, or by hashing their value), we can detect if two code fragments are *isomorphic*. Code which is isomorphic can be detected if we can see a 1-to-1 relationship between identifiers and constants.

Take, for example, the following two code fragments.

```
int maxofthree(int x, int y, int z)
 {
   if ((x>y) && (x>z)) return(x);
   if (y>z) return(y);
   return(z);
 }
int bigtriple(int b, int a, int c)
 {
   if ((b>a) && (b>c)) return(b);
   if (a>c) return(a);
   return(c);
 }
```

The variable names have been changed, yet the algorithm is identical. If we record the order of occurrence of each identifier in each fragment, we can check to see if there is a 1-to-1 correspondence between them.

Here, we have

Identifier	**Tag**		**Tag**	**Identifierq**
x	id1	⇔	id1	b
y	id2	⇔	id2	a
z	id3	⇔	id3	c

Note that there must be a 1-to-1 correspondence in both directions. If a new identifier `q` in the first fragment is introduced which appears to correspond to `b`, then this ends the similarity between the fragments as `b` already corresponds to `x`.

The second version of Ctcompare introduced the enumeration of identifiers, the encoding of numeric constants and code isomorphism. To speed up the comparison, groups of 4 tokens were joined to make 32-bit integers, and integer comparisons were used to reduce the cost of token comparison. Unfortunately, this only took effect once the beginning of a matching “string” was found; byte-by-byte tokens were required to find the beginning of a match. The initial isomorphism code was buggy and complicated; the actual solution turned out to be very elegant.

9. The Rabin-Karp Comparison Algorithm

While the second version of Ctcompare met four of the five design requirements outlined in Section5, it was too slow. The code had to perform an O(n×m) search to find the beginning of a matching “string”, and then had to compare consecutive tokens to determine the length of the matching run.

In practice, runs of matching tokens below a certain length can be discounted. If 2 consecutive tokens are considered a match, then there will be a large number of `if (` and `while (` matches. Structural similarity between two C program fragments becomes significant around 20 tokens; therefore, runs shorter than 20 tokens can be ignored.

This threshold for significant similarity can be used to improve the program’s performance. A colleague pointed me at the Rabin-Karp Algorithm[2] which is ideally suited to this situation.

With Rabin-Karp, we do not try to find the beginning of a matching token “string” if there is a similarity threshold *m*. Instead, we start at the beginning of the two input streams, and calculate hashes of the first two token runs of size *m*. If the hashes match, then we have found a possible matching token run. If the match fails, we shift by 1 token in the second input stream and repeat the process. The essential ingredient here is a rolling hash function that is O(1) to shift down 1 token, e.g. for the input string “carousel”, calculating hash(“arous”) from hash(“carou”) is easy.

The third implementation of Ctcompare uses Rabin-Karp to find *potential* code matches; we ignore identifier hash values and numeric constant values here for speed. This produces a set of possible code matches. This is not excessively bad, as Rabin-Karp will produce hash collisions anyway.

Once a possible code match is found, it is then passed to the isomorphic comparison test to find possibly *longer* runs of matching tokens, or to disprove the ‘match’ found by Rabin-Karp.

We also keep track of matches that have been found, so that we don’t report smaller matches in the same area, e.g. “HELLO” matches “HELLO”, but we can ignore the fact that “ELO” matches “ELO”. This makes the third version of

Ctcompare about 8 to 16 times faster than the brute-force approach.

10. Validating the Lexical Approach

In the USL vs. BSDi court case in the 1990s, USL alleged the existence of significant amounts of 32V code in the Net/2 distribution from the University of California, Berkeley, which had been released under a BSD license. Kirk McKusick filed a deposition in the case in which he stated:

I have found only 56 lines of code in five kernel files that appear to match lines of code in 32V. These 56 lines are out of the total of 539 source files and 230,995 lines of source code in the Net2 kernel. The files in which I have found matches are discussed below. …

There are 358 lines of text in the Net2 `ufs/disksubr.c` *file. Conservatively, fourteen of these 358 lines of source code in* `ufs/disksubr.c` *are the same as lines in* `sys/dsort.c` *from 32V. …*

There are 697 lines of text in the Net2 `ufs/ufs_inode.c` *file. Being conservative, nine of these 697 lines of source code in* `ufs/ufs_inode.c` *are the same as lines in* `sys/iget.c` *from 32V. …*

There are 592 lines of text in the Net2 `kern/subr_prf.c` *file. Generously, four of the 592 lines of source code in* `kern/subr_prf.c` *are the same as lines in* `sys/prf.c` *from 32V. …*

There are 403 lines of text in the Net2 `kern/kern_exit.c` *file. Only three of the 403 lines of source code in* `kern/kern_exit.c` *are the same as lines in \texttt{sys/sys1.c}. …*

There are 1863 lines of text in the Net2 `ufs/ufs_vnops.c` *file. Being generous, 26 of these 1863 lines of text in* `ufs/ufs_vnops.c` *match code in 32V.[6]*

The third version of Ctcompare finds all but 7 of these lines: most of the missing lines are singles or doubles that fall below the threshold of 20 tokens. The total run time on a 2GHz Pentium is 50 seconds. However, the comparison finds several other runs of similar code which were not found by McKusick:

Net/2

```
if (bswlist.b_flags & B_WANTED) {
  bswlist.b_flags &= ~B_WANTED;
  thread_wakeup((int)&bswlist);
}
```

32V

```
if (bfreelist.b_flags&B_WANTED) {
  bfreelist.b_flags &= ~B_WANTED;
  wakeup((caddr_t)&bfreelist);
}
```

11. Final Comments on Ctcompare

The development of Ctcompare follows the paradigm: Write it to work, then write it to work correctly, then write it to work correctly and fast. The lexical approach chosen is a good balance between the fast line-by-line comparison and a more awkward fully-semantic analysis. The Rabin-Karp algorithm is superb at finding possible matches, which are then filtered through the isomorphic comparison: the combination of the two is quite elegant. To preserve proprietary code confidentiality, names of identifiers and the full value of numeric constants are not revealed; at the same time, enough context is provided to significantly reduce the number of false positives found. Regardless of all the above, any comparison of millions of lines of code will still be O($n \times m$) and thus slow.

To date, no comprehensive comparison of Linux versus System V or UnixWare source code has been conducted by Ctcompare, as I have yet to find someone who has access to the latter and who will send me a token stream. Token streams for some early versions of System V have been provided by members of the Unix Heritage Society[11]. While they have not revealed any UNIX code in Linux, the token streams have been compared with earlier version of the UNIX source code. This has revealed that several of the AUSAM modifications to UNIX[4] done in the 1970s by UNSW were still in System V in the early 1990s.

Ctcompare version 1.3 is available at `http://minnie.tuhs.org/Programs/`. It includes a collection of tokenised source trees, including several System V releases. Overall, it consists of 1,000 lines of C, 100 lines of header files and 250 lines of *lex* source.

References

[1] **A. Bouard**, *Interview with Darl McBride*, Nov, 2003, http://www.01net.com/article/220196.html

4. AUSAM was a modification to 6th Edition UNIX done at UNSW in the 1970s to make the system more robust and to support more simultaneous users.

[2] **D. Ellard**, *The Rabin-Karp Algorithm*, Jul, 1997, http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node43.html

[3] **G. J. Kirsch**}, *The Changing Roles of Patent and Copyright Protection for Software*, Apr, 2000, http://www.gigalaw.com/articles/2000-all/kirsch-2000-04-all.html

[4] **G. Lehey**, *SCO's evidence of copying between Linux and UnixWare*, Jan, 2004, http://www.lemis.com/grog/SCO/code-comparison.html

[5] **D. Marti**, *SCO to Reveal Allegedly Copied Code*, May, 2003, http://www.linuxjournal.com/article.php?sid=6877}

[6] **M. K. McKusick**, *SECOND DECLARATION OF DR. KIRK MCKUSICK IN SUPPORT OF THE REGENTS OF THE UNIVERSITY OF CALIFORNIA'S AMICUS BRIEF RE MOTION FOR PRELIMINARY INJUNCTION*, Jan, 1993, http://minnie.tuhs.org/UnixTree/Net2Kern/930119.mckusick.decl.2

[7] **B. Perens**, *Analysis of Linux Code that SCO Alleges Is In Violation Of Their Copyright and Trade Secrets*, Aug, 2003, http://www.perens.com/SCO/SCOCopiedCode.html

[8] **E. S. Raymond**, *Comparator and Filterator*, Sept, 2003, http://www.catb.org/~esr/comparator/

[9] **E. S. Raymond**, *SCO's Evidence: This Smoking Gun Fizzles Out*, Aug, 2003, http://www.catb.org/~esr/writings/smoking-fizzle.html

[10] **The SCO Group**, *Original Complaint in TSG vs IBM*, Mar, 2003, http://sco.tuxrocks.com/Docs/IBM/complaint3.06.03.html

[11] **W. Toomey**, *The Unix Heritage Society*, http://www.tuhs.org

[12] *Definition of Lexeme*, On-line Computing Dictionary, Apr, 1996, http://www.instantweb.com/foldoc/foldoc.cgi?lexeme