Comparing C Code Trees

Dr. Warren Toomey
Faculty of IT, Bond Uni
wtoomey@staff.bond.edu.au

This presentation describes a tool to lexically compare two C code trees for possible code copying, analyses 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.

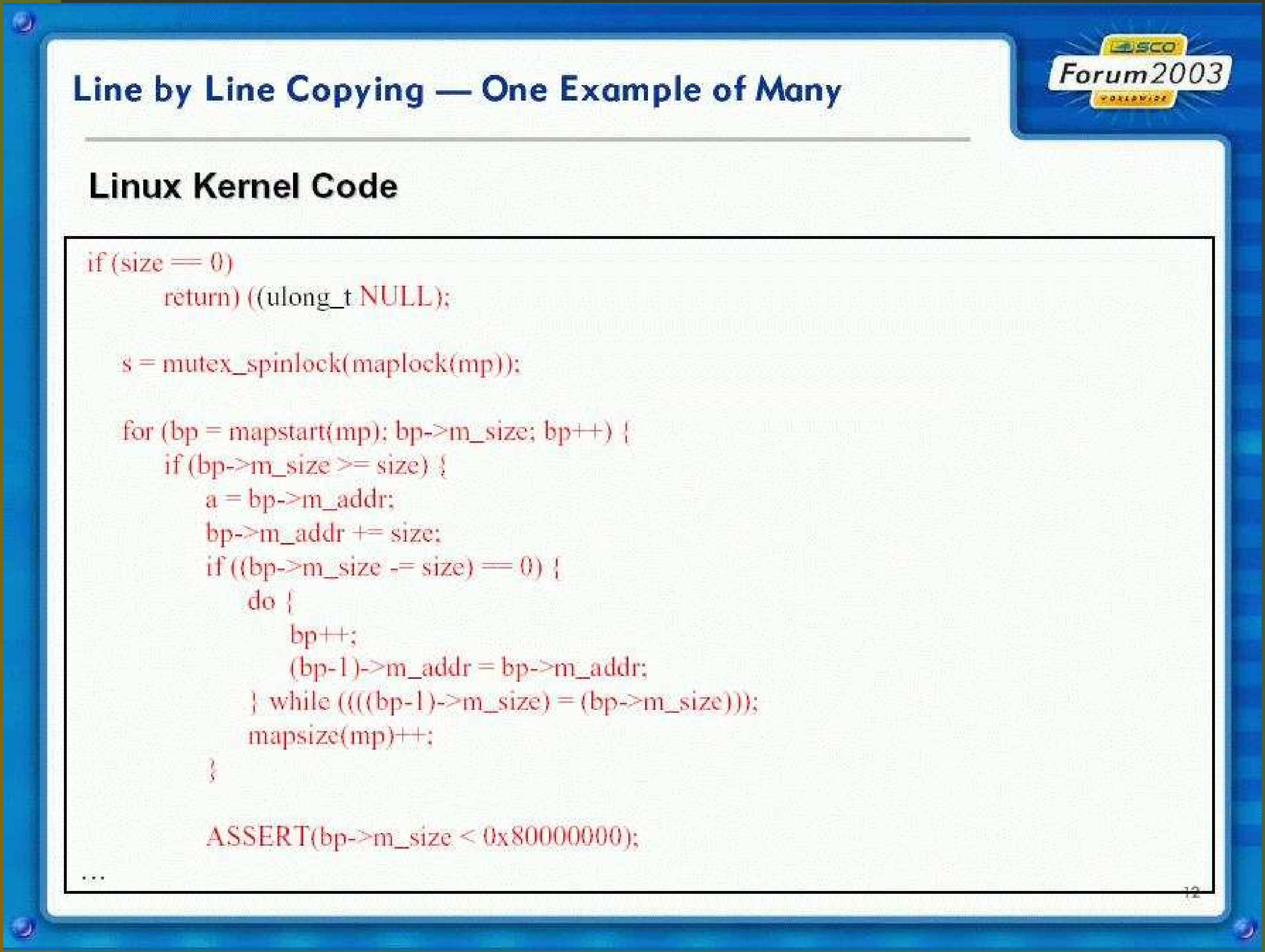
Why Write Such a Tool?

- To detect student plagiarism.
- To determine if your codebase is ‘infected’.
- To trace code geneaology, useful for the new field of computing history.
- To confirm/deny FUD:

  Sontag specifically claimed that there is “significant [SCO] copyrighted and trade secret code within Linux ... It’s all over the place”. Code has been “munged around solely for the purpose of hiding the authorship or origin of the code”.

  Linux Journal, May 2003.

SCO's Evidence of Stolen Code



Line by Line Copying — One Example of Many

Forum2003

Linux Kernel Code

```
if (size == 0)
        return) ((ulong_t NULL);

    s = mutex_spinlock(maplock(mp));

    for (bp = mapstart(mp); bp->m_size; bp++) {
        if (bp->m_size >= size) {
            a = bp->m_addr;
            bp->m_addr += size;
            if ((bp->m_size -= size) == 0) {
                do {
                    bp++;
                    (bp-1)->m_addr = bp->m_addr;
                } while ((((bp-1)->m_size) = (bp->m_size)));
                mapsize(mp)++;
            }

            ASSERT(bp->m_size < 0x80000000);
...
```

12

SCO's Evidence of Stolen Code

Corresponding System V code:

```
s = splimp();           /* enter critical region */
for (bp = mp; bp->m_size && ((bp-mp)< MAPSIZ); bp++) {
  if (bp->m_size >= size) {
    a = bp->m_addr;
    bp->m_addr += size;
    if ((bp->m_size -= size) == 0) {
      do {
        bp++;
        (bp - 1)->m_addr = bp->m_addr;
      } while ((bp - 1)->m_size = bp->m_size);
    }
    splx(s);            /* exit critical region */
    return (a);
```

Comments on This Example

- Yes, SysV code was placed into the Linux kernel. Not by IBM, but by SGI for the ia64 platform.
- The code was removed due to its “ugliness”.
- Code first appeared in UNIX in 1973, and is based on a 1968 algorithm by Knuth.
- Code was published in book form in 1997.
- Caldera *released* early UNIXes under a BSD license in 2002, *before* the code was added to Linux!
- The Unix Heritage Society (which I run) was critical in tracing the code’s genealogy.
- I had to send SCO copies of the UNIX code from before System V!

The Other Code Presented by SCO

- SCO presented another snippet of code in Linux which it claimed was from System V.
- The code turns out to be Berkeley Packet Filter code, written in 1991 and placed under the BSD license:

  Copyright (c) 1990, 1991 The Regents of the University of California. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer...

- It seems that when the BPF code was integrated into System V, the BSD copyright notice was removed.
- Thus, if SCO believes that this is their code, someone violated the BSD license between 1991 and now.

Issues with Code Comparison

- Can rearrangement of code be detected?
  - per line? per sub-line?
- Can “munging of code” be detected?
  - variable/function/struct renaming?
- What if one or both codebases are proprietary? How can third parties verify any comparison?
- Can a timely comparison be done?
- What is the rate of false positives?
  - of missed matches?

Code Comparison Requirements

- Must permit the detection of code rearrangement to some extent.
- Must be reasonably fast: $O(n^2)$ or better.
- Any code representation must be exportable without giving the original code away; this allows other to verify any code comparison.
  - e.g. Eric Raymond's *comparator* hashes code lines, and then compares the hashes.
- If possible, it should detect different coding of the same algorithm: renamed variables, constants, functions etc.

My Idea: Lexical Comparison

- Break C code into lexical *tokens*, compare runs of tokens.
- This removes all semantics, but deals with code rearrangement.
- C has about 100 lexical units:

  Single chars: [ ] { } + - * / % !

  Multiple chars: ++ && += !=

  Keywords: int char return if for while do break

  Values: identifiers “strings” ‘x’ 1000L
- Encode each token into 1 byte, then do “string” comparison on 2 strings, one for each code tree.

1st Implementation: Brute-Force

- 1st implementation was a proof of concept one.
- For each token in first string: find matching strings starting at this point in the second string:

  HELLOTHEREHOWAREYOU?
  WHATCELLOBEWARELOTHERE?
- Values of identifiers, string & other constants are stripped, so as to not reveal original code.
- Brute-force is $O(n * m)$, n & m are string lengths.
- Slowed down by keeping "LINE" tokens within the data structures, and by poor loop design.

1st Implementation: Poor Accuracy

- Missed some matches due false skipping, e.g:

  HELLOTHEREHOWAREYOU?
  WHATCELLOBEWARELOTHERE?

  (H)ELLO matches (C)ELLO, but can't skip to THERE, as (L)LOTHERE matches (E)LOTHERE.

- Too many false matches due to loss of identifiers, and also common C features:

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

and

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

Fixes to 1st Version

- Remove skipping. Add run-time switch to ignore C pre-processor directives.
- Encode bottom 16-bits of numeric constants.
  - Allows rejection of non-matching numeric constants.
  - Reveals some details of the original code, not enough to breach copyright issues (I hope).
- Still many false positives, e.g:

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

and

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

Code Isomorphism

- Code that is isomorphic can be detected if we can see a 1-to-1 relationship between identifiers:

```
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);
  }
```

Code Isomorphism

- Must record order of occurrence of each identifier in each file: 1st id, 2nd id, 3rd id, 1st id again, 3rd id again.
- Then check 1-to-1 identifier correspondence:

Identifier	Tag		Tag	Identifier
x	id1	$\Leftrightarrow$	id1	b
y	id2	$\Leftrightarrow$	id2	a
z	id3	$\Leftrightarrow$	id3	c

- But if new identifier $q \Rightarrow b$, error as $x \Leftarrow b$.

2nd Version: Isomorph + Hashes

- Keep 16 bits of numeric constants. Keep *enumerated* identifier tags. Still allows export of tokenised code tree.
- Isomorphic code reduces false matches.
- Hash groups of 4 tokens into 32-bit integer.
- Integer compares reduce cost of comparison, but only once the start of a run is found. Still, about 4x faster than brute-force.
- Must still traverse token by token to find start of run.
- Initial isomorphism code was buggy and complicated; actual solution turned out to be very elegant.

Rabin-Karp Comparison Algorithm

- Existing code: search to find start, search to find matching run.
- Assume we want to find minimum match of m tokens.
- Once possible start of a run is found, must do up to m token comparisons to prove match, i.e. strcmp("cat", "car").
- Instead, calculate *hash* of first token run of size m, *hash* of second run of size m, compare hashes, i.e. hash("cat") == hash("car")?
- Use rolling hash function that is $O(1)$ to shift 1 token, i.e. calculating hash("art") from hash("car") is easy, if we have string "*the cartridge*".

3rd Version: Use R-K for Speed

- Given minimum threshold m, use Rabin-Karp to find matching token hashes of length m from first code tree in the second code tree.
- No identifier tags nor numeric values used here for speed; more non-matches than matches.
- Possible hash collisions anyway, so then follow up with isomorphic test to find possibly *longer* runs, or disprove the 'match' found by Rabin-Karp.
- Keep track of matches, so we don't report smaller matches in the same area, e.g. "HELLO" matches "HELLO", but "ELO" matches "ELO".
- About 8 to 16 times faster than the brute-force approach.

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 Net/2, which had been released under a BSD license.
- Kirk McKusick's deposition in the case: there are only 56 lines of code common to the 32V and Net/2 kernels (13K lines in 32V, 230K in Net/2).
- Lexical comparison finds all but 7 of these lines: singles or doubles below the threshold of 20 tokens. Total run time on 2GHz Pentium: 50 seconds.
- However, the comparison finds several other runs of similar code not found by McKusick.

32V cf. Net/2: Missed Matches

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);
}
```

Comments on Implementation

- The brute force version works but is slow.
- Algorithms are like tools: use them where you can. Know a repertoire, and have a good reference book.
- Why did I consider lexical analysis? I was exposed to a compiler course.
- Isomorphic comparison: elegant & clever IMHO.
- Latest version hashes identifier tags: allows export of tokenised code tree. Shows if identifiers are the same without revealing whole code tree.
- Regardless, any comparison of millions of lines of code will be slow, as it is $O(n^2)$.

Where to get the Implementation?

- For info on SCO vs. IBM, see `http://www.groklaw.com`
- My lexical comparison tool is at `http://minnie.tuhs.org/Programs/`
- It includes a collection of tokenised source trees, including several System V releases.
- Overall: 1,000 lines of C, 100 lines of header files, 250 lines of *lex* source.
- Eric Raymond's line-based comparison tool is at `http://www.catb.org/~esr/comparator/`
- His also has several heuristics built in, so each tool should validate the other.