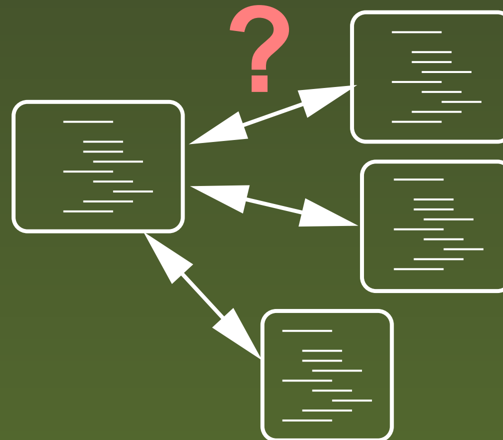
Comparing C Code Trees, 2.2

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

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

Ctcompare 2.2, a Work in Progress

- Back in 2004 I published a paper on a code tree comparison tool, ctcompare 1.2.
- It allowed two trees of source code to be compared for possible code copying.
- This is a work in progress to extend the tool, to compare one code tree against multiple other code trees simultaneously.



Why Write Such a Tool?

- To detect student plagiarism.
- To determine if your codebase is ‘infected’:
  - by proprietary code from elsewhere, or
  - by open-source code covered by a license like the GPL.
- To trace code genealogy between trees separated by time (e.g. versions),
  - useful for the new field of computing history.

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 divulging the original code.
  - this allows others to verify any code comparison.
  - however, *something* of the original code's structure has to be divulged.
- If possible, it should detect different coding of the same algorithm: renamed variables, constants, functions etc.

Original Idea: Lexical Comparison

- Break the C code in each tree into lexical *tokens*, then compare runs of tokens between trees.
- This removes all the code's semantics, but does deal with code rearrangement (but not code "munging").
- C has about 100 lexical units:

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

  Multiple chars: ++ && += !=

  Keywords: int char return if for while do break

  Values: identifiers "strings" 'x' 1000L labels
- We encode each token into 1 byte, then do "string" comparison on 2 strings, one for each code tree.

CTF Files: Serialised Token Streams

- Step 1 is to parse all code files in a tree, and produce a single file containing the serialised token stream for all code files in the tree.
- Each token is 1 byte.
- Tokens with “values” (e.g. identifiers) are followed by a 2-byte hash of the value.
- Filenames are embedded, as are 1-byte LINE tokens, so that line numbers can be inferred.

```
for ( a = 5 ; a < sum ; a ++ )
```

Advantages of CTF Representation

- The source code is not released, but the code's essential logic is released.
- Variable names, constant values etc. are not revealed.
  - However, if two files use the same variable name, they will have the same hash value.
- Code rearrangement via whitespace can be detected.
- Rearrangement of code blocks can be detected.
- CTF files can thus be exported to third parties without revealing the original code.

CTF Example

```
385: do {
386:   id891 = id64003 [ id100 ] ;
387:   id64003 [ id100 ] = NUM48 ;
388:   if ( id891 > NUM408 )
389:     id891 = NUM426 ; else
390:   if ( id891 > NUM446 )
391:     id891 = NUM446 ; else
392:     id891 = NUM48 ;
393:   id55378 -> id32607 = id891 ;
394:   id55378 += NUM49 ;
395:   id32068 ( id100 ++ ) ;
396: } while ( id100 < ( id31677 + NUM445 ) ) ;
```

First Lexical Comparison Approach

```
foreach (token in one CTF file)
{
   walk the other CTF file to find
     a matching run of tokens;
}
```

```
if (a<2) if (b<17) c = a * b; a++; printf( ..

while (x!=5) if (b<17) c = a * x; b++; ...
```

- $O(M * N)$, where M, N are # of tokens in each file.
- Algorithms (e.g. Rabin-Karp) used for performance.
- Cannot compare a CTF file to a set of CTF files.

How Much is Significant Copying?

- Common code snippets are not significant:

Snippet	Token Length
for (i=0; i<val; i++)	13 tokens
if (x>max) max=x;	10 tokens
err= stat(file, &sb); if (err)	14 tokens

- **Axiom:** runs of <16 tokens are not significant.

- **Idea:** use a run of 16 tokens as a lookup key to find other trees with that same run of tokens.

New Approach in 2.2

- Take a group of 16 tokens (excluding in-line values), known as a *tuple*.
- Use the tuple as the primary key in a database search.
- The search result is a list of all files in all code trees which have that run of tokens.
- Implication: for a CTF file of N tokens, it will generate N-15 keys. Database will be big.
- However, key collisions should be rare.
- A key collision (>1 CTF file using that key) indicates potential code copying.

Example Key and Result

Key
23 5f 40 5f 29 7b 5f 28 5f 23 5f 29 3b 5f 23 5f
-> id != id) { id (id -> id) ; id -> id

Result				
CTF File	**Offset**	**Name Offset**	**Line #**	**Last Line #**
32V	219c	1e7c (dev/du.c)	92	94
Net/2	81de	7759 (kern/ktrace.c)	299	301

The Tuple Database, Quantified

- Four unrelated code trees, approx. 100K LOC.
- Gdbm database size: 41 Mbytes. # of tuples: 350K.

% Tuples with N Results					
1	82%	3	3%	5	0.7%
2	11%	4	1.6%	6	0.5%

- 6,059 tuple collisions between CTF files, i.e. 1.7%.
- Worst offending tuples:

NUM,NUM,NUM ...	2,745	"str","str"...	1,809
,NUM,NUM,...	2,730	,"str","str",...	1,764

New Lexical Comparison Approach

```
foreach (token in one CTF file)
{
  calculate 16-tuple key starting here;
  get list of results from database;
  foreach result (not from our CTF file)
  {
    perform real token + value comparison
      between both files, determine actual
      run length of common tokens+values;
  }
}
```

Eureka!!

- After writing the last two slides, I realised I was wrong.
- It is wasteful to traverse the CTF file: 82% of 16-token tuples are unique.
- In fact, the database already holds a list of all potential code similarities:
  - 6,059 tuple collisions between different CTF files out of 350K tuples.
- We just need to iterate over the keys with multiple results from different files.
- Then, perform a real token + value comparison for all result combinations.

Algorithm Revisited

```
foreach (key in database)
{
  get result: list of files with this key;
  skip keys with no multi-file result collisions;
  for (all result combinations)
  {
    perform real token + value comparison
      between both files, determine actual
      run length of common tokens+values;
  }
}
```

Performance: Old vs. New

- Ctcompare 1.2 was $O(N * M)$
  - where N, M are # tokens in each CTF file.
  - time to compare the 32V tree vs. Net/2 tree: 73s.
- Ctcompare 2.2 seems to be $O(N)$ or $O(N\, log_2 N)$
  - where N is # tuple collisions in the database.
- Time to compare the 32V tree vs. the Net/2 tree: 2s
- Time to compare 4 unrelated code trees: 5.6s
- However: CTF insert into database is slow.
  - This needs to be improved.
- Later, I will examine the $O()$ complexity properly.

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

- We do this when evaluating actual matching run lengths.
- We must record the 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 $x \Leftarrow b$.

Isomorphism Example: 32V cf. Net/2

Net/2 22 tokens long

```
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);
}
```

Code Isomorphism

- Code isomorphism is only used to compare identifiers.
- Other tokens with values (numeric literals, string literals, character literals, labels) are compared exactly.
- The idea of isomorphism sounds complicated. The actual solution turned out to be very elegant.
- However, it does slow things down somewhat:
  - 15 seconds vs. 2 seconds for the 32V cf. Net/2 tree comparison.
  - 7,405 matches vs. 36 matches: most are bogus.

Some Optimisations

- Set a run-time limit on the number of isomorphic relations permitted.
- This helps to prevent false positive runs like this:
  - *int creat(); int dup(); int exec(); . . .* versus
  - *int getlogin(); int setlogin(); int sysacct() . . .;*
- Extend the “16 tokens in a run” as the database key:
  - XOR together the 16-bit hashes for the literal values (not identifiers).
  - Append to 16-byte token run for an 18-byte key.
  - This reduces the size of the “large result” keys (e.g. NUM,NUM,NUM,NUM, ...) by 3/4.
- Both improve the performance, too.

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 16 tokens.
- However, the comparison finds several other runs of similar code not found by McKusick.
  - e.g the isomorphic example a few slides back.

Comments on the Work in Progress

- Lexical analysis into tokens + values is a big win.
  - exportable CTF files, comparison on code structure, also finds code rearrangement
- Isomorphic code comparison is also a big win.
- Use of token tuples as keys was serendipitous.
  - trades space for time, but also reduces algorithm compexity
  - allows for simultaneous comparison between multiple code trees
- Proof of concept works, now need to tidy up, refine the solution & analyse it.