Cobra:  
Fast Structural Code Checking  

Gerard J. Holzmann  
*Nimble Research*  
SPIN 2017
when it really *has* to work...
tool-based code review @ jpl

Nightly Build Log
~3K compiler calls

Static Code Analysis for
Defect Detection &
Coding Rule Compliance Checking

~2.8 M lines of C

gcc –Wall –pedantic
coverity
codesonar
semmle
uno

code review
database
‘scrub’

total analysis time
~15hrs
now consider this scenario

• an in-flight anomaly occurs
  o manual analysis reveals the cause:
    o a function passes an array argument of the wrong size
      • function expects an array of 16 elements
      • the call passes an array of 8 elements
      • data corruption results
  
• Q: can this happen anywhere else in the code?
  o write new checkers for some of the static analyzers
  o wait 15 hours for the cumulative check to be completed...
  o meanwhile, on a few million miles away.....
so, why is the analysis taking so long?

2.8 M lines of code — — > pre-processing

lexical analysis

building AST

building CFG

parsing

symbol table

alias analysis

a lot of time is spent here

a dumb idea: what if we skipped all that?

run all checkers

our new check runs

hours later....

and plug-in new checkers here.....

our new checker
A linked list of lexical tokens with annotations
(token types, ranges, levels of nesting for parentheses, brackets, and braces, etc.)
now we can do pattern matching on tokens

• first the obvious:  
  \$ cobra -cpp -e j *.ch
  
  # compare with: grep -e j *.ch
  
  # differences:
  
  # we match tokens, not characters
  
  # code is optionally pre-processed

• match on token types:
  
  \$ cobra -e @qualifier *.ch
  
  # e.g., const, volatile

  \$ cobra -e @modifier *.ch
  
  # e.g., long, short, unsigned

• matching regular expressions:
  
  \$ cobra -e '{ .* malloc ^free }' *.ch
  
  # functions calling malloc, but not free
using sets and ranges

```bash
$ cobra -c "m switch; n {; c top no default; d" *.[ch]
```

Example query: match `switch` statements without a `default` clause

```c
switch (f->n->ntyp):
    case UNLESS:
        attach_escape(f->sub->this, e);
        break;
    case IF:
        for (z = f->sub; z; z = z->nxt)
            attach_escape(z->this, e);
        break;
    case D_STEP:
        /* attach only to the guard stmt */
        escape_el(f->n->sl->this->frst, e);
        break;
    case ATOMIC:
    case NON_ATOMIC:
        /* attach to all stmts */
        attach_escape(f->n->sl->this, e);
        break;
}
```

- `m[ark]` defines a set
- `n[ext]` moves the match position forward
- `{ ... }` is a predefined range that we can query
context: querying code

- interactive queries (sets, ranges)
- regular expression pattern matching
- inline programs

Source Code \(\rightarrow\) cobra \(\rightarrow\) Patterns of Interest

simple parallel query processing

\(N\) CPU cores

1 2 ... \(N\)
command scripts and creating sets
example: are function calls to g reachable from f?

```python
def find(f, g):
    fcts # mark function definitions (predefined)
    n {
    c g # move mark to start of function body
    c g # does it contain tokens named g?
    b \( ; \) b # move mark back to function name
    \[ >1 \] # save names of these functions in set 1
    r # reset
    fcg f # mark functions reachable from f (predefined)
    \[ <&1 \] # intersect new marks with set 1
end

find(server, malloc) # can malloc be called once server starts?
display # display any matching function names
```
predefined operations on sets

>n  # save   current marks in set n
<n  # assign: replace current marks with set n
<|n # union: add marks from set n to current
<&n # intersect: keep only marks also in set n
<^n # subtract: keep only marks not in set n

n is 1..3
two additional sets are used internally for storing the current and the previous set of marks (allowing a fast ‘undo’ on all operations)
variable binding

- find assignments to the control variable of a for-loop, inside the loop body:

  \[
  \$ \text{cobra -e "for } (x:@ident .*) \{ .*(:x)=.* \} " *.c
  \]

- find local variable declarations that aren’t used in the function body:

  \[
  \$ \text{cobra -e "} \{ .*@type(x:ident ^:x* } " *.c
  \]

  to avoid matching on structure declarations
implementation of the matching algorithm

Regular Expression Matching Can Be Simple And Fast
(but is slow in Java, Perl, PHP, Python, Ruby, ...)

Russ Cox
rsc@swtch.com
January 2007

Introduction

This is a tale of two approaches to regular expression matching. One of them is in widespread use in the standard interpreters for many languages, including Perl. The other is used only in a few places, notably most implementations of awk and grep. The two approaches have wildly different performance characteristics:

![Graphs comparing Perl 5.8.7 and Thompson NFA](image-url)
ken thompson’s algorithm
example: find expressions with multiple side-effects

\$ \text{cobra -e ‘( \text{-}\text{-}\text{|}\text{+}\text{-}\text{+} ) ^;*( \text{-}\text{-}\text{|}\text{+}\text{-}\text{+} )’} \ast .c$

sml_dsa.c:
17: 213 \hspace{1em} 1A1 = (\text{ulong}) (*p++)*(q--);

Thompson’s algorithm
(CACM 11:6 1968)
inline Cobra programs

inline programs are by default executed once for each token in the input stream

example: a recursive function

```
$ cobra some_file.c

:%{
  function fact(n)
  {  if (n <= 1)
   {    return 1;
   }
   return n*fact(n-1);
  }

  print "10! = " fact(10) "\n";
  Stop;
%

```
writing inline Cobra programs

- inline programs can access (and modify) all token attributes, use associative arrays, recursive functions, etc.

<table>
<thead>
<tr>
<th>strings:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.fct</td>
<td># function name</td>
</tr>
<tr>
<td>.fnm</td>
<td># file name</td>
</tr>
<tr>
<td>.txt</td>
<td># token text</td>
</tr>
<tr>
<td>.typ</td>
<td># token type</td>
</tr>
</tbody>
</table>
| .fct           | # name of containing function, or "global"

<table>
<thead>
<tr>
<th>numbers:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.round</td>
<td># nesting level of ()</td>
</tr>
<tr>
<td>.bracket</td>
<td># nesting level of []</td>
</tr>
<tr>
<td>.curly</td>
<td># nesting level of {}</td>
</tr>
<tr>
<td>.len</td>
<td># length of token text</td>
</tr>
<tr>
<td>.lnr</td>
<td># linenumber</td>
</tr>
<tr>
<td>.mark</td>
<td># user-definable integer value</td>
</tr>
<tr>
<td>.seq</td>
<td># token sequence number</td>
</tr>
<tr>
<td>.range</td>
<td># the nr lines in the associated range</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tokens:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.nxt</td>
<td># the immediately following token</td>
</tr>
<tr>
<td>.prv</td>
<td># the immediately preceding token</td>
</tr>
<tr>
<td>.jmp</td>
<td># move to other end of range, eg from { to } or back</td>
</tr>
<tr>
<td>.bound</td>
<td># link to bound symbol reference</td>
</tr>
</tbody>
</table>
quite simple inline programs can already do useful things

```%
%{ # check the identifier length for all tokens
    # and remember the longest in a variable q
    if (@ident && .len > q.len){ q = .; }
}%

    # the scan over all tokens is now completed,
    # q retains its value, which can now be printed:

%{
    print "longest ident is: " q.txt " = " q.len " chars\n";
    Stop; # stops after the above line is printed
}%```
associative arrays & external commands
e.g., find the most common token-type trigrams

```c
{ q = .nxt; r = q.nxt; if (.typ != "" && q.typ != "" && r.typ != ")
    Trigram[.typ, q.typ, r.typ]++;
}
%
track start _tmp_
%
if (cpu != 0)
    Stop;
}
A unify(0);
for (i in Trigram)
    print i.txt "t" sum(Trigram[i.txt]) "\n";
}
Stop;
%
track stop
!
sort -k2 -n _tmp_ | tail -10; rm -f _tmp_
```

- associative array
- divert output to a temporary file
- post-process with cpu 0 only
- collect data from all cores at cpu 0

shell escape

the 10 most frequently occurring type trigrams

```
$ cobra -f play/trigram *.ch
ident,oper,chr 209
const_int,oper,ident 231
oper,oper,ident 232
storage,type,oper 239
key,const_int,oper 250
storage,type,ident 702
ident,oper,const_int 1000
type,oper,ident 1298
oper,ident,oper 3541
ident,oper,ident 5695
```
finding uninitialized variable use
(a flow-sensitive property)

```
$ cd Unix/V7/usr/src/cmd
$ cobra -f(dfs_uninit *.c

...  
cat.c:16 declaration of dev

cat.c:50 uninitialized use
...  
```

(1) the script creates links to capture
a rudimentary control-flow graph
for each function (if/else/goto)

(2) using recursive fct calls, it then
performs a DFS over the CFG to
find suspicious execution paths

```
main(argc, argv)
char **argv;
{
    ...
    int dev, ino = -1;
    struct stat statb;
    setbuf(stdout, stdbuf);
    ...
    statb.st_mode &= S_IFMT;
    if (statb.st_mode!=S_IFCHR && statb.st_mode!=S_IFBLK) {
        dev = statb.st_dev;
        ino = statb.st_ino;
    }
    ...
    while (--argc > 0) {
        ...
        if (statb.st_dev==dev && statb.st_ino==ino) {
            fprintf(stderr, "cat: input %s is output\n", 
                fflg?t":*: argv);
            fclose(fi);
            continue;
        }
    }
    ...
}
```

159 .c files, 30 KLOC, 1 core, 0.8 seconds
5 accurate warnings + 1 false positive
so does it scale? multi-core processing

18,633,817 Lines of Code of the Linux 4.3 distribution, with 39,144 .c and .h files

checking 2 types of queries:
- find empty else stmts
- find all switch stmts without default clause

using 1..32 CPU cores

4 cores or more: query processing < 1 sec.
documentation, manuals, downloads: www.spinroot.com/cobra

thanks!