interactive code checking with Cobra

a Tutorial

Gerard Holzmann
Nimble Research
gholzmann@acm.org

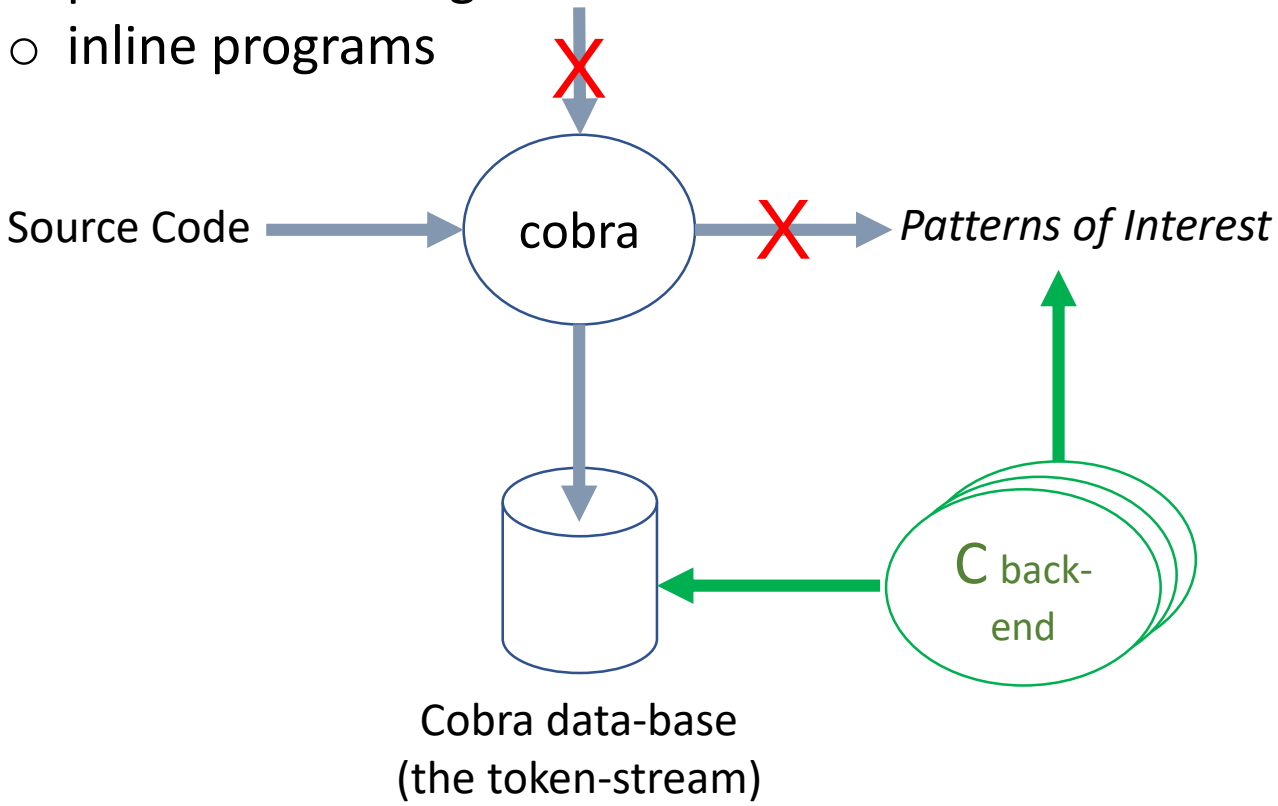
topics covered

1. *background* and principle of operation
   - installation and configuration
   - guide to online documentation
2. *pattern* queries and regular expressions
   - exercises
3. *interactive* queries
   - token attributes
   - sets and ranges
   - functions
   - reading files, libraries
   - exercises
4. *scripted* queries
   - recursive functions
   - associative arrays
   - the query libraries
   - using concurrency
   - exercises
5. *standalone* checkers
   - using concurrency: multi-threaded checkers
6. use of Cobra for *runtime verification*
   - using live data or event-logs

context

we can link checkers written in C to Cobra's front-end

- interactive query commands over sets & ranges
- pattern matching commands
- inline programs



the front-end creates the token-stream and can then hand off control to a user-defined back-end that can perform its own processing, using the full power of C

the back-end code can be multi-threaded to scan the token-stream in parallel, just like the built-in commands

standalone checkers

linked to the cobra front-end

example checkers are included in the distribution in the $Cobra/src_app subdirectory, including checkers for a range of cwe properties, which are all defined as multi-threaded checkers – they look like this:

```
 1   #include "c_api.h"
 2
 3   typedef struct Names Names;
 4   struct Names {
 5           char *nm;
 6           int  cnt;
 7           Names *nxt;
 8   } *names;
 9
10  int
11  newname(char *s)
12  {
            ...
26  }
27
```

```
28  void
29  cobra_main(void)                // the interface point
30  {
31    for (cur = prim; cur; NEXT)   // main loop over the token sequence
32    { if (TYPE("ident"))          // if (strcmp(cur->typ, "ident") == 0)
33      { if (verbose)
34        { printf("n_%d ", newname(cobra_txt()));
35        } else
36        {    printf("ident ");
37        }
38      } else
39      {    printf("%s ", cur.txt);
40      }
41      if (MATCH(";")              // if (strcmp(cur->txt, ";") == 0)
42      ||  MATCH("}")
43      ||  TYPE("cpp"))
44      {       printf("\n");
45      }
46    }
47  }
```

the Cobra API definition

header-file, library, and utility functions

three files define the Cobra API:

c_api.h — a header file with data definitions and function prototypes, which defines the key token structure (typedef Prim), and token navigation functions

c.ar — a library with the Cobra front-end code, from these source files:
cobra_lex.o, cobra_prep.o, cobra_prim.o, cobra_heap.o and cobra_links.o

cwe_util.c — some utility functions for setting up multi-threading and for using associative arrays; example functions are:

```
clear_marks(from, to)
run_threads(void *(*f)(void*)),
store_var(…)
is_stored(…)
unstore_var(…)
```

compiling and linking a checker

checker template

```
#include "c_api.h"

void
cobra_main(void)  // called by the front-end after preprocessing is complete
{
      for (cur = prim; cur; cobra_nxt())  // the main loop over all tokens
      {   // do stuff to/with cur, the current token
      }
}
```

the backend code is compiled by linking to library file c.ar,
and if multi-threaded with –pthread, as in:

```
$ cc –O2  –I. –std=c99  –o yourfile  yourfile.c  c.ar  -pthread
```

predefined standalone checkers

the multi-threaded cwe checkers in $COBRA/src_app
and the precompiled binary in $COBRA/bin_...

```
$ ls -l src_app/cwe*
-rw-r--r--+ 1 gh None  2587 Nov 29  2018 cwe.c
-rw-r--r--+ 1 gh None  1018 May  6 13:39 cwe.h
-rw-r--r--+ 1 gh None 12089 Apr 26 13:10 cwe_119.c
-rw-r--r--+ 1 gh None  9236 Apr 27 09:20 cwe_120.c
-rw-r--r--+ 1 gh None  5291 Mar 14 13:44 cwe_131.c
-rw-r--r--+ 1 gh None  3450 Apr 27 10:26 cwe_134.c
-rw-r--r--+ 1 gh None  4105 Mar 14 13:44 cwe_170.c
-rw-r--r--+ 1 gh None  6435 Mar 14 13:54 cwe_197.c
-rw-r--r--+ 1 gh None  8216 May  6 13:30 cwe_416.c
-rw-r--r--+ 1 gh None 10934 May  6 13:30 cwe_457.c
-rw-r--r--+ 1 gh None  1467 Mar  8 14:12 cwe_468.c
-rw-r--r--+ 1 gh None  4423 Mar  8 14:53 cwe_805.c
-rw-r--r--+ 1 gh None  6335 May  6 13:35 cwe_util.c
$ ls -l  bin_cygwin/cwe*
-rwxr-xr-x+ 1 USER None 271152 Jun  4 09:21 ../bin_cygwin/cwe.exe
```

for comparison:
cobra scripted equivalents
for each cwe check are also available
in $COBRA/rules/cwe/...

multi-threading

setting up a run

Consider the examples in the cwe_....c files with multi-threaded versions of a selection of CWE checks, and observe the following:

- all thread-local data is collected in a single data-structure, e.g. (in cwe_119.c):

```
typedef struct ThreadLocal119 ThreadLocal119;
struct ThreadLocal119 {
    Prim *sol, *eol;
    TrackVar *ixvar, *tested, *modified, *suspect, *params;
};
```

- at the start of a run for a specific check, we call an …_init() function to setup the multi-threading
- the threads are initiated with a call to utility function run_threads(fct)
  - each thread function is setup to process only the range of tokens that is passed to it by the interface routine
- at the end of a run we call a …_report() function to collect, possibly combine, and report the data gathered by each thread
- to avoid conflicts, each thread declares its own copy of a token reference function Prim *mycur, rather than using the predefined token reference

```
extern Prim *cur
```

running standalone checkers

passing arguments

- The Cobra front-end interprets all arguments that it recognizes, so if you want to pass additional arguments to your checker they must be distinct from the existing ones.
- Usually the original Cobra parameters suffice, but when they don't:
  - the only way to pass new arguments requires the checker-name to include the substring "cwe" – which causes unrecognized arguments to be made available in a shared global text string called "cwe_args"
- The processing that makes this happen can be found in
  src/cobra_prep.c:1192-1215
  and an example of the parsing of the result can be found in
  src_app/cwe.c:79-84
- Error handling of arguments that are *not* recognized by either the front-end or the back-end checkers can be a bit challenging

exercise (advanced)

build a simple checker

1. Build, compile, link, and run a checker that counts the number of semi-colons in its own input source

2. For extra points: try to make it multi-threaded and collect the data from the different threads before reporting it