Automated Code Repair to Ensure Memory Safety – 2018-07-16 Meeting

2018-07-16

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM18-0853

Automated Code Repair (ACR) session

Topics:

- Using ROSE in the SEI line-funded projected “Automated Code Repair to Ensure Memory Safety”
- Other uses of ROSE by the Secure Coding team
- Difficulties in building ROSE
  - Spack
- maybe more topics?

Goals and strategy

Goal: Repair code (both C source code and x86 binary) to enable a proof of memory safety.

This entails formal reasoning in regards to:

1. The repaired program is memory-safe.
2. The repaired program is equivalent (modulo undefined behavior) to the original program.

Strategy:

1. Translate original source code to a simple intermediate representation (IR), annotating the IR with tags that record how to convert it back to original code.
2. Disassemble binary executables/libraries (using Pharos) to the same IR.
3. Repair the IR (whole-program analysis, operating on output files from steps 1 and 2 above).
4. Convert repaired IR back to the original code as closely as possible.

Quick summary of progress so far

A quick-and-dirty prototype has been written in Python, using a simple C99 parser. We are using this prototype to iterate on ideas for:
(1) static analysis and (2) repair with reconstruction of the original AST.

This tool does not yet support reconstruction of the original source code. (It doesn't see the source code until it has already been preprocessed.)

Previous work (FY16): We have used ROSE for automated repair of integer overflows that lead to buffer overflows.

(We noticed that the option "`-rose:unparse_tokens`" sometimes leads to crashes. This might be already fixed in newer version of ROSE.)

Design for IR↔source mapping

- We augment the IR with *tags* that record how to transform the IR code back to source.
  - Tags do not affect the semantics of the IR.
  - The nodes of the original abstract syntax tree (AST) are tagged with the exact text of corresponding source code (including whitespace and macros).
- We have developed a set of reversible transformations that start with the original AST and transform it step-by-step to the IR.
- The IR is repaired and then transformed back to source using the tags.
  - If a repair invalidates a tag, then the tag is ignored.
- IR-to-source transformation should be sound (semantics-preserving).
- **Notice any problems?**

Sequence Points

An IR usually has a well-defined evaluation order, but C does not.

- E.g.:

```
int x = 0;
x = x++;
printf("x=%i\n", x);
```

- What value is printed?
  - With gcc 4.6, "x=1" is printed.
  - With gcc 5.2, "x=0" is printed.

Therefore, naively reconstructing C expressions from totally-ordered IR instructions is unsound in general.

We solve this problem by introducing IR instructions that explicitly indicate a partial order for execution.

IR instructions to indicate unsequenced ops

```
     spawn_unseq [L1, ..., Ln], end;
L1:;
     body1;
     merge_unseq end;
L2:;
     body2;
     merge_unseq end;
     ...
Ln:;
     bodyn;
     merge_unseq end;
end:;
```

Semantics: for all $i \neq k$, every instruction in $body_i$ is unsequenced w.r.t. every instruction in $body_k$.

IR instructions to indicate weak sequencing

```
    weak_seq mid, end;
    body1;
    merge_weak end;
mid:;
    body2;
end:;
```

Semantics: side effects in $body_1$ are unsequenced w.r.t. $body_2$, and value computations in $body_1$ are sequenced before $body_2$.

Note: There are also other IR instructions needed to indicate other types of partial order in C.

Repair that breaks IR->AST

```
1.  int foo() {
2.      ...
3.      m = malloc(1000);
4.      if (!m) goto fail;
5.      aux1(m);
6.      for (int i = 0; m[i] != 0; i++) {
7.          aux2(m[i]);
8.      }
9.      ...
10.     free(m);
11.     return 0;
12. fail:
13.     ...
14.     return -1;
15. }
```

Repair that breaks IR->AST

```
1.  int foo() {
2.      ...
3.      m = malloc(1000);
4.      if (!m) goto fail;
5.      aux1(m);
6.      for (int i = 0; 1; i++) {
7.          if (!(0 <= i)) {goto fail;}
8.          if (!(i < 1000)) {goto fail;}
9.          if (!(m[i] != 0)) {break;}
10.         ok = aux2(m[i]);
11.         if (!ok) {goto fail;}
12.     }
13.     ...
14.     free(m);
15.     return 0;
16. fail:
17.     ...
18.     return -1;
19. }
```

Corner case in C (but not C++)

```
1.  #include <stdio.h>
2.  int main() {
3.      int again = 1;
4.      {
5.          int y = 0;
6.  foo:
7.          printf("y = %p:%i\n", &y, y);
8.      } /* Note: y implicitly becomes undefined here.
               IR has an explicit de-initialization instruction.  */
9.      if (again) {
10.         again = 0;
11.         int z = 99;
12.         printf("z = %p:%i\n", &z, z);
13.         goto foo;
14.     }
15.     return 0;
16. }
```