ACR MemSafety Notes

Presenter: Will Klieber

Date: May 30, 2018

Document markings

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.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

DM18-0717

What about false positives from static analysis?

Reducing false positives is difficult, but we don't need to.

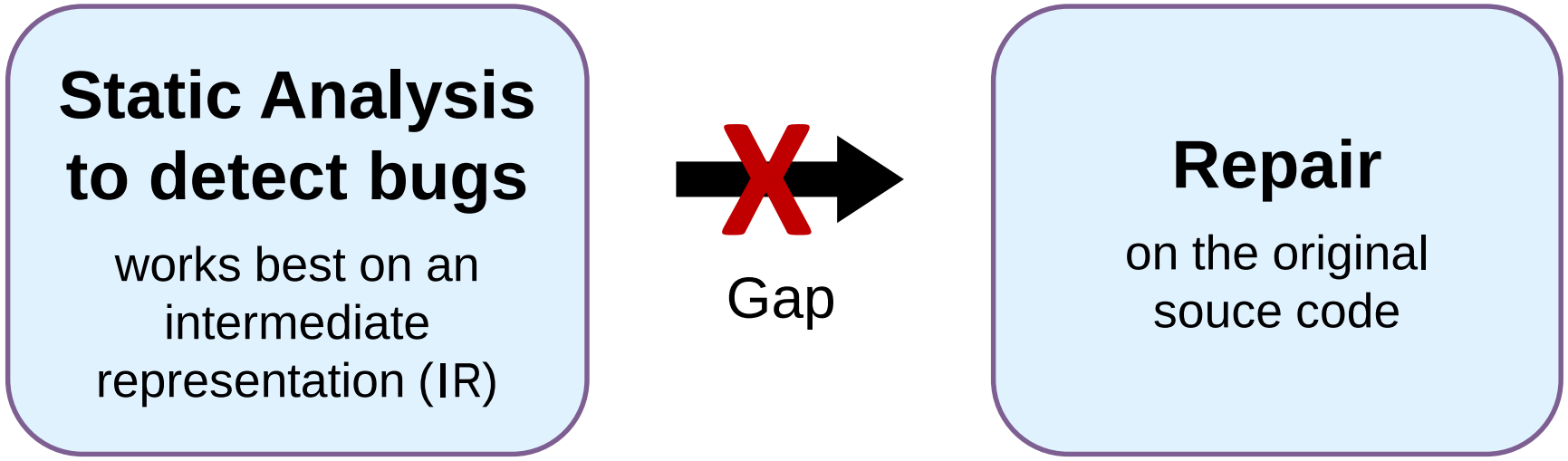
It's okay to 'repair' false positives, if we don't break code.

- The small runtime overhead is often acceptable.

Why at source code level (instead of a compiler pass)?

Repair on source code	Compiler pass
One-time change to source code.	Permanent change to the build process.
Repairs can be easily manually audited.	Must trust the tool.

Gap between static analysis and repair of source code



Difficulty: must map repaired IR back to source.

Design for IR↔source mapping

- We augment the IR with *tags* that record how to transform the IR back to source.
  - Tags do not affect the semantics of the IR.
- 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 must be sound (semantics-preserving).

Example of AST <-> IR

Original source code:

```
2.     if (*p) {
3.         p++;
4.     }
```

Intermediate Representation (IR)

```
  temp_vars [t01, t02, t03, t04, t05];
  ; @[('stmt_start',), ('line', 2)]
  ; @[('if_stmt', 't01', 'L_if_jump_1', 'L_if_done_4')]
  ; @[('unary_op', 't05', 'L_done_unary_10')]
  t05 = p;
  t01 = *t05;
L_done_unary_10:;
L_if_jump_1:;
  if (t01) goto L_if_true_2 else goto L_if_false_3;
L_if_true_2:;
L_8:; @[('scope', 'L_scope_end_9')]

  ; @(('begin_expr_stmt', 'L_end_stmt_5'), ('line', 3))
  compound_assign L_end_unary_6; @[('postfix_var_incr',)]
  t04 = p;
  p = t04 + 1;
  t02 = t04;
L_end_unary_6:; @[('expr_stmt',)]
L_end_stmt_5:;
L_scope_end_9:;
  goto L_if_done_4;
L_if_false_3:; @[('omit',)]
L_if_done_4:;
```

What is memory safety?

- To prove memory safety, we first need a precise definition.
- A program is **memory-safe** iff, on every possible execution of the program:
  - every memory access (read or write) is to a location in a currently allocated region, and
  - every value passed to `free(·)` is the start of a live region returned by `malloc(·)`.
- Possible executions include those where the compiler leaves **gaps of unallocated memory** between variables on a stack frame. (Padding in `structs` is not a gap of unallocated memory.)
- A **proof of memory safety** is often divided into two parts:
  1. **Spatial:** Writing or reading beyond the bounds of a memory region. (FY18+ work)
  2. **Temporal:** Writing or reading to a region after it has been deallocated. (FY19+ work)
- Dereferencing NULL is a memory violation, but low severity (only denial-of-service). Plus, an automatic repair would often be of little value. So we ignore.

Spatial Memory Safety

- Heuristic: If a program performs arithmetic on a memory address *p1* to obtain a new memory address *p2*, and *p2* is later dereferenced, then *p2* should be in the same allocated memory region as *p1*.
  - The C standard actually requires this (on pain of undefined behavior) for arithmetic on values of pointer type. (However, pointers can be casted from/to integers. And, from the binary side, there is no such requirement.)
  - This heuristic is conservative in the sense that it catches all spatial memory violations, but it might have false positives.
- To check the bounds of a pointer dereference, we do a backwards analysis to find the possible memory allocations from which the pointer was derived.
  - More detail on the next slide.

Preconditions to ensure memory access is within bounds

- Consider a program in an intermediate representation.
- A *trace* is a sequence of program states corresponding to an execution of the program.
- Given a pure (side-effect-free) expression *e* and a state *s*, we write "$e|_s$" to denote the value of *e* in state *s*.

- Given a program variable *x*, a trace t, and a step i in the trace t, if the value of *x* (at step i in t) was computed by pointer arithmetic on the result of a memory allocation (malloc or stack allocation), then:
  - let MemLo(*x*)$|_{t[i]}$ denote the lower bound of this memory region (inclusive),
  - let MemHi(*x*)$|_{t[i]}$ denote the upper bound of this memory region (exclusive).
- For each memory access *p*, we generate a precondition ensuring it is within bounds: MemLo(p) ≤ p < MemHi(p).

Ensuring preconditions are satisified

- From previous slide: For each memory access *p, we generate a precondition ensuring it is within bounds: φ = “MemLo(p) ≤ p < MemHi(p)”.
- Now our goal is to either:
  - Prove that φ is always satisfied, or
  - Insert a runtime bounds check, by constructing a formula ψ such that:
    - ψ always has the same truth value as φ (at the given program point under the background theory of program semantics), and
    - ψ is representable using only the variables defined at the given program point (e.g., no “MemLo” or “MemHi”).
- To accomplish this goal, we will use backward and forward analysis, relating pre-/post-conditions of adjacent program points.

Pre-/post-conditions of a statement

- For each predecessor p of an IR statement c in the control-flow graph, let $jc_{p \to c}$ denote the condition under which control flows from p to c.
- With our IR, $jc_{p \to c}$ will be either:
  1. the constant `True`,
  2. a simple variable (for the `True` branch of a conditional jump), or
  3. the negation of a simple variable (for the `False` branch).
- "Simple variable": local, non-`static`, non-`volatile`, non-aliased.
- Conditional jumps in our IR have the form:
  "`if (`v`) goto` L_T`; else goto` L_F`;`" where v is a simple variable.

Pre-/post-conditions of a statement

- For each predecessor p of an IR statement c in the control-flow graph, let $jc_{p \to c}$ denote the condition under which control flows from p to c.
- If a boolean formula ϕ is a precondition of c, then the formula $jc_{p \to c} \Rightarrow \phi$ must be a postcondition of p. Symbolically:

$$\phi \in pre(c) \;\Rightarrow\; \text{“}jc_{p \to c} \Rightarrow \phi\text{”} \in post(p)$$

- In the reverse direction:

$$\left(\bigwedge_{p \in preds(c)} \text{“}jc_{p \to c} \Rightarrow \phi\text{”} \in post(p) \right) \quad \Rightarrow \quad \phi \in pre(c)$$

Pre-/post-conditions of a statement

- For each predecessor p of an IR statement c in the control-flow graph, let $jc_{p \to c}$ denote the condition under which control flows from p to c.
- If a boolean formula ϕ is a precondition of c, then the formula $jc_{p \to c} \Rightarrow \phi$ must be a postcondition of p. Symbolically:

$$\phi \in pre(c) \;\Rightarrow\; \text{“}jc_{p \to c} \Rightarrow \phi\text{”} \in post(p)$$

- In the reverse direction:

$$\left(\bigwedge_{p \in preds(c)} \text{“}jc_{p \to c} \Rightarrow \phi\text{”} \in post(p) \right) \quad \Rightarrow \quad \phi \in pre(c)$$

- What if some predecessors are unreachable (dead code)?

Assignment statement

- **Notation for Substitution**. We write "$\phi[x \rightarrow e]$" to denote the result of substituting all occurrences of x with e in ϕ.

- For a statement s of the form "$v_L := e_R$", where v_L is a simple variable and e_R is a pure expression (free of side effects):

$$\phi[v_L \rightarrow e_R] \in pre(c) \quad \Leftrightarrow \quad \phi \in post(c)$$

- Example: For a statement c of the form "`x := 1;`":

$$\text{“}1 < 2\text{”} \in pre(c) \quad \Leftrightarrow \quad \text{“}x < 2\text{”} \in post(c)$$

Allocation

- Notation: given an IR statement s, "$\{i\}@s < n$" means that "$i < n$" will be true the next time control passes through s (after the operation of s).
- For a statement s of the form "v_L `:= malloc(`v_{size}`)`", where v_L and v_{size} are simple variables:

$$\phi[MemLo(v_L) \rightarrow \{v_L\}@c] \in pre(c) \quad \Leftrightarrow \quad \phi \in post(c)$$
$$\phi[MemHi(v_L) \rightarrow \{v_L\}@c + v_{size}] \in pre(c) \quad \Leftrightarrow \quad \phi \in post(c)$$

Widening

- When propagating pre-/post-conditions, we record its history (where it originated from and what program points it has already passed through).
- To ensure termination of the static analysis, we don't propagate a pre-/post-condition formula through the same program point more than once.