FY2018-20 LSI Project

Automated Code Repair (ACR) to Ensure Memory Safety

Presenter: Will Klieber

Date: Jan 29, 2019

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.

DM19-0085

Automated Code Repair (ACR) for Memory Safety

Problem: Software vulnerabilities constitute a major threat to DoD.

- Memory violations are among the most common and most severe types of vulnerabilities.
- Static analysis helps find bugs, but the volume of alerts is often overwhelming.
- Huge amount of code is in use by DoD, with unknown number of security vulnerabilities.

Solution: Repair code to enable proof of mem safety.

Approach:

- Transform source code to an intermediate representation (IR).
- Try to prove that each memory access is within bounds ("spatial memory safety") and not to a deallocated region ("temporal memory safety").
- If unable to prove, repair code so that proof succeeds.
- Map the transformed IR back to source code.

Quick summary of project status

- Recent progress:
  - Design for handling multiple build configurations.
  - Fat pointers: replace pointers of type `T*` with objects of type: `struct {T* ptr; T* base; size_t len;}`.
  - Mapping from the abstract syntax tree (AST) to the original source-code text.
- Next steps:
  - Finish the fat-pointer implementation.
  - Finish the Source<->AST implementation, tie it in with existing AST<->IR implementation.
- Expected year-end accomplishments:
  - Paper on soundly repairing source code to use fat pointers in the presense of multiple build configurations.

What about false positives from static analysis?

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

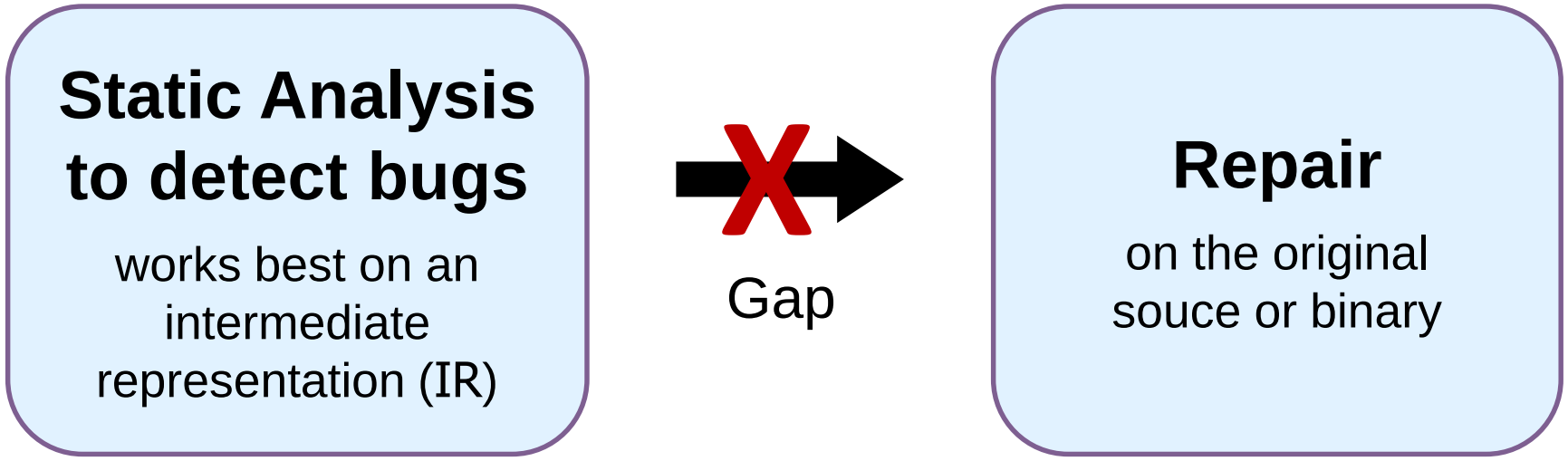
It's okay to 'repair' false positives, if we don't break code.

- The small runtime overhead is often acceptable.

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

Repair on source code	Compiler pass
Repairs can be easily manually audited.	Must trust the tool.
Repairs can be manually tweaked to improve performance.	Difficult to remediate performance issues caused by repair.
Change to the source code.	Change to the build process.

Gap between static analysis and repair of source code



Difficulty: must map repaired IR back to source or binary.

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 **abstract syntax tree (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:;
```

Multiple build configurations

Definition: a *configuration* is a (partial) assignment of values to symbols used in conditional preprocessor directives (`#ifdef`, `#ifndef`, `#if`, and `#elif`).

There are two main situations in which a source file might need be repaired so that it is correct under multiple configurations:

- The codebase has compile-time options for things such as targeting a specific platform or including/excluding certain features (e.g., FIPS-compliant mode for OpenSSL).
- A single header file is `#included` multiple times in a single build, but with different configurations selected by the including file. A common use of this is to achieve (in C) some of the functionality provided by templates in C++.

Syntactically ill-formed fragments inside #ifdefs

Snippet from coreutils regexec.c, lines 3549--3554:

```
#ifdef RE_ENABLE_I18N
        if (dfa->mb_cur_max > 1)
            bitset_merge (accepts, dfa->sb_char);
        else
#endif
            bitset_set_all (accepts);
```

Multiple build configurations

Given a configuration C and an original source code file F, we write $Repair(C, F)$ to denote the result of repairing F under configuration C.

Algorithm 1:

1. Repair each configuration individually.
2. Try to merge all the repairs at the level of (unpreprocessed) source code.
3. Verify that, for each config C, the merged file reduces under C to $Repair(C, F)$.

If Algorithm 1 fails, we will refactor the preprocessor directives using the approach described in “Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell” (TSE 2017) and repeat Algorithm 1 on the refactored source code.

Get experience with real-world codebases before further developing algorithm.

Source ↔ AST

- The Source ↔ AST mapping is used to translate repairs at the AST level to repairs on the original source-code text.
- The C preprocessor is the main source of difficulty.
  - For repairs to macro uses, we replace the macro with its repaired expansion.
  - For repairs to **`#include`**d code, we repair the **`#include`**d file.
  - Our memory-safety repairs never delete code. If they did, more care would be needed with respect to **`#ifdef`** blocks.
- Considerations of whitespace (e.g., indenting inserted code) is also a concern.
- We are finishing implementation of a modification to Clang to extract a Source<->AST mapping. We also plan to do this in Rose as well.

Fat Pointers – struct definition and allocation wrapper

Those pointers that can safely be converted to fat pointers are called *fat-eligible*.

For each pointer type *T** with a fat-eligible occurrence, we introduce a new struct definition (below) and an allocation wrapper (to the right):

```
struct FatPtr_T {
    T*      ptr;
    T*      base;
    size_t  len;
};
typedef struct FatPtr_T FatPtr_T;
```

```
static inline FatPtr_T
fatcalloc_T(size_t size) {
    FatPtr_T ret;
    ret.ptr  = malloc(size);
    ret.base = ret.ptr;
    ret.len  = size;
    if (!ret.ptr) {ret.size=0;}
    return ret;
}
```

Modifications to source code for fat-pointer repair

Fat-eligible variables and fields are changed as follows:

- Declarations are changed from type *T** to `FatPtr_`*T*.
- Allocations are changed to use the wrappers.
- A dereference **x* is changed to **x*`.ptr` and a bound check is prepended:
  `if (!(`*x*`.base <= `*x*`.ptr && `*x*`.ptr < `*x*`.base + `*x*`.len)) {abort();}`
- When passed to a standard-library function, *x* is changed to *x*`.ptr` and a bound check is inserted before.
- Subtraction of pointers (*p* - *q*) is changed to substraction of the ptr fields of fat pointers (*p*`.ptr - `*q*`.ptr`). Comparison operators are handled the same way. Addition and substraction of integers to pointers is handled similarly.
- The function `realloc(...)` is changed to never re-use the same starting memory location when increasing the allocation size.

Eligibility for Fat Pointers

Definition: An ***allocation site*** is a call site (in the AST or IR) of memory allocator.

Definition: A function is ***external*** iff its source code is not available to our analysis.

Definition: An allocation site *c* is ***fat-eligible*** unless any of the below are true:

- An external function gets passed a pointer from which memory allocated at *c* is (transitively) reachable
- The bytes of memory allocated at *c* are accessed by a pointer other than a pointer-to-pointer type.
- The allocation site *c* is a `mmap` that might be backed by a file.
- A variable that may point to fat-ineligible memory also may point to mem allocated at *c*.

To determine fat-eligibility, we use a field-sensitive, flow-insensitive, context-insensitive analysis. (Some context sensitivity can be provided heuristically by inlining.)

A variable or field is fat-eligible if every mem region it can point to is fat-eligible.

Related Work

- Todd Austin et al (PLDI 1994) developed a source-to-binary compiler pass that inserts bounds checks using fat pointers. It is not binary-compatible with existing libraries.
- SoftBound (PLDI 2009) likewise inserts bounds checks via a compiler pass. It stores the bounds in a separate region of memory, allowing interfacing with existing libraries.
  On a subset of the SPEC 2006 benchmarks, slowdown was 436% (Keaton 2014).
- CETS (ISMM 2010): Compiler-Enforced Temporal Safety. Associates unique label with each allocated object. Tracks by pointer, not pointed-to object (unliked DANGNULL) – this can cause false positives on pointer arithmetic that erroneously abort the program.
- Shaw, Dugget, & Hafiz (2014) [OpenRefactory]: limited source transformation to use fat pointers for strings, as well as other opportunistic repairs. Can't prove mem safety.
- STONESOUP: Some binary-repair work was done to mitigate memory violations, but it doesn't extend to **proving** memory safety.
- Gregory Duck and Roland Yap, "Heap bounds protection with low fat pointers" (2016): restricts the possible allocation sizes to a fixed finite set and allocates same-sized objects in one region of virtual address space. Requires 64-bit pointers and mem overcommit. Works on Linux.

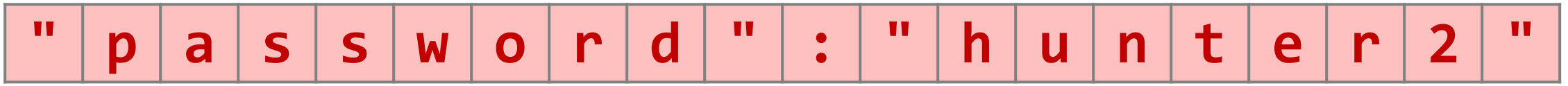
Additional Details

Temporal memory safety

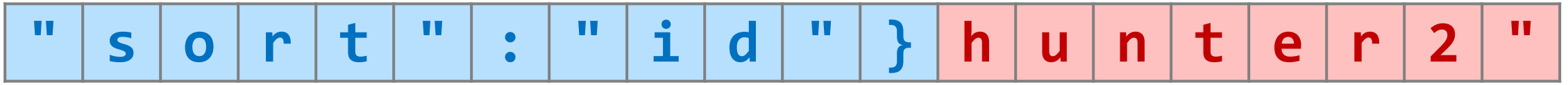
- We plan to use a technique similar to DANGNULL (Lee et al., NDSS 2015).
- DANGNULL works as follows: for each heap-allocated object, DANGNULL maintains a collection of which other heap objects point to it (or point inside it).
- When an object is freed, **all dangling pointers to it are zeroed out**.
- DANGNULL had a runtime overhead of 80% on the SPEC2006 benchmarks.
- DANGNULL operates as runtime enforcement, with very lightweight static analysis (as opposed to trying to repair only code that is determined to be potentially buggy).
- We plan to add a more thorough static analysis to try to prove that a freed object does not have any dangling pointers to it.
  - If the proof succeeds for a given object, then we do not insert the expensive corresponding run-time instrumentation code because it is no longer needed.
  - For example, for every doubly-linked list node allocated at a particular allocation site, we might be able to prove that the only heap pointers to the node are its two neighboring nodes.

Leakage of sensitive information in re-used buffer

Buffer contents after **first HTTP request**:



Buffer contents after **second HTTP request** (from a different client):



Upper bound for reading: most recently written location

Sub-object problem

Sometimes a pointer to a sub-object inside a larger object should be constrained to the sub-object, as in the snippet below.

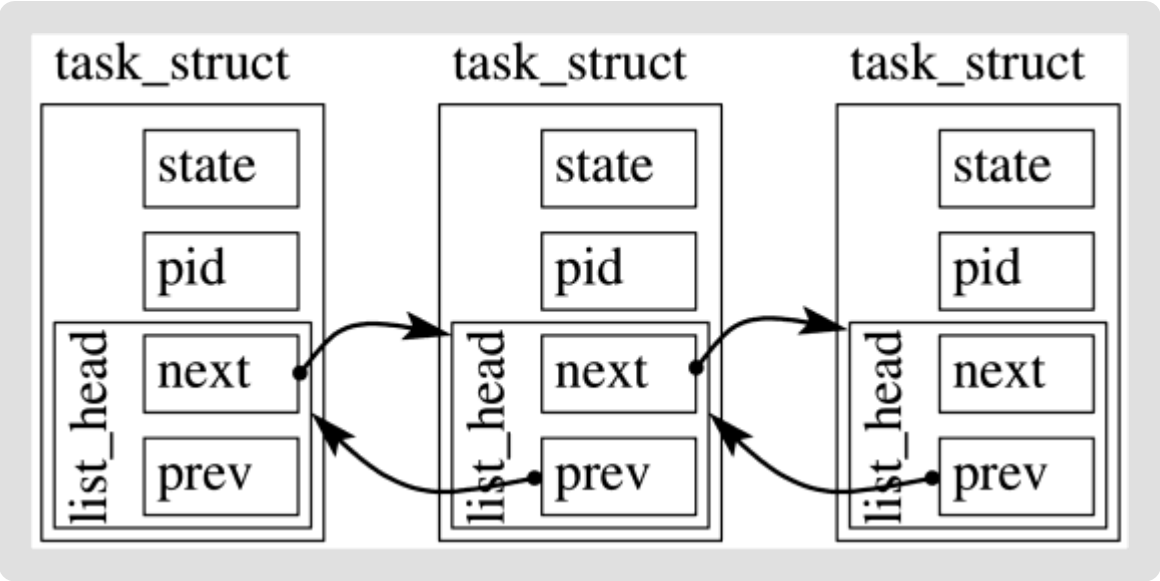
Best upper bound of `acct->id` is `acct->id + sizeof(char[8])`,
not `acct->id + sizeof(bank_acct)`.

```
struct bank_acct {
    char id[8];
    int balance;
};
...
bank_acct* acct = malloc(sizeof(struct bank_acct));
strcpy(acct->id, "overflow...");
```

Linked List sub-object in Linux kernel

```
struct list_head {
  list_head *next;
  list_head *prev;
};

struct task_struct {
  long state;
  pid_t pid;
  list_head sibling;
};
```




```
#define list_entry(ptr, type, member) \
  ((type *)((char *)(ptr) - offsetof(type,member)))

task_struct *cur_task, *next_task;
...
next_task = list_entry(cur_task->sibling.next, task_struct, sibling);
```