Research Review 2019

Automated Code Repair (ACR) to Ensure Memory Safety

Will Klieber

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

[Distribution Statement A] Approved for public release and unlimited distribution.

Copyright 2019 Carnegie Mellon University.

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.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[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.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM19-0928

RESEARCH REVIEW 2019

Automated Code Repair (ACR) to Ensure Memory Safety

Will Klieber

[Distribution Statement A] Approved for public release and unlimited distribution.

Automated Code Repair (ACR) for Memory Safety

Software vulnerabilities constitute a major threat to DoD.

- Memory violations are among the most common and most severe types of vulnerabilities.
  - 15% of CVEs in the NIST NVD and 24% of critical-severity CVEs.
  - iPhone iOS CVE-2019-7287 (exploited by Chinese government?)
  - Android Stagefright (2015)
  - CloudBleed (2017)
- Huge volume of code is in use by DoD, with unknown number of vulnerabilities.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Solution: Automated Code Repair

Automatically repair source code to assure memory safety.

- Abort program immediately before a memory violation would occur.
- Attackers no longer able to exploit the program.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Approach to Automated Code Repair

- Transform source code to an intermediate representation (IR), retaining mapping.
- Try to assure that each memory access is within bounds (spatial memory safety) and not to a deallocated region (temporal memory safety)
- If unable to assure, repair code to ensure memory safety.
  - Use ***fat pointers*** to store bounds information when possible.
- Map the transformed IR back to source code.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Automated Code Repair (ACR) tool as a black box

Input: Buildable codebase

Output: Repaired source code, suitable for committing to repository

We plan to support C and have limited support for C++

ACR Tool

Envisioned use of tool

- Intended for ordinary developers
- Use before every release build
- Use occasionally for debugging builds

© 2019 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Inserting missing bounds checks

When the result of pointer arithmetic is used to access memory, ensure that it is still within the bounds of the original allocated region.

```
v = calloc(n, sizeof(int));
...
scanf("%i", &k);
for (i=0; i < k; i++) {
   if (v[i] < 0) break;
   v[i] += 1;
}
```

Repair: Insert bounds check

```
if (i >= n) abort();
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

What about false positives from static analysis?

Reducing the false-positive rate is difficult.

But we don’t need to reduce it!

It’s okay to “repair” false positives as long as we don’t break the code.

The small runtime overhead is often acceptable.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

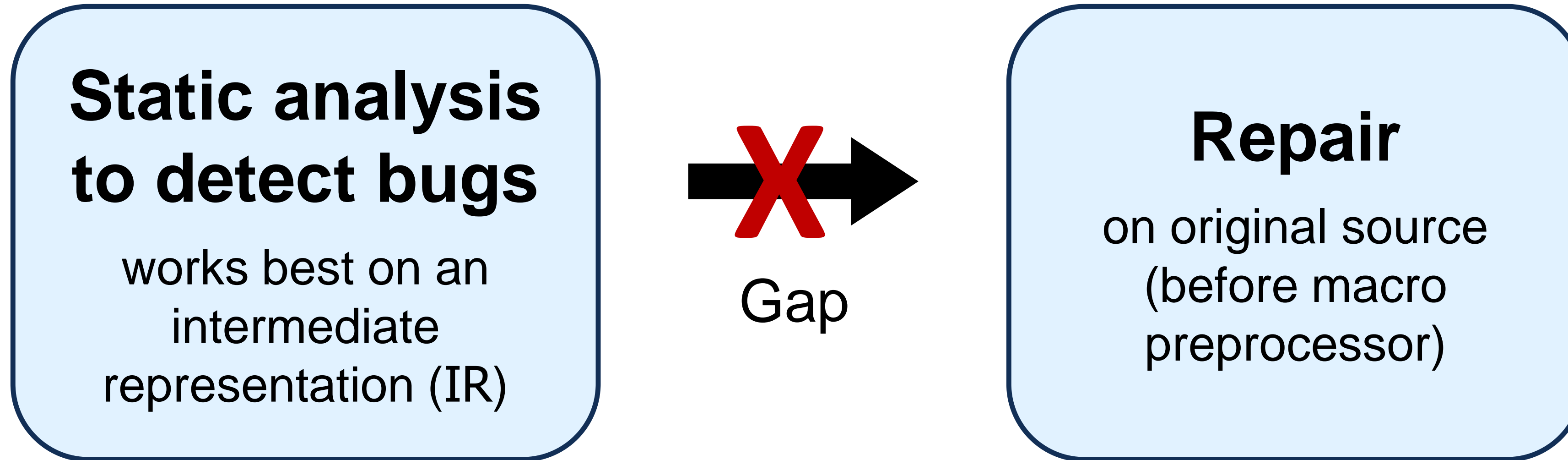
Why repair of source code instead of as a compiler pass?

Repair of source code	Repair as a 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.
Changes to source code are frequent and easily handled.	Changes to the build process may be more difficult and error-prone.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

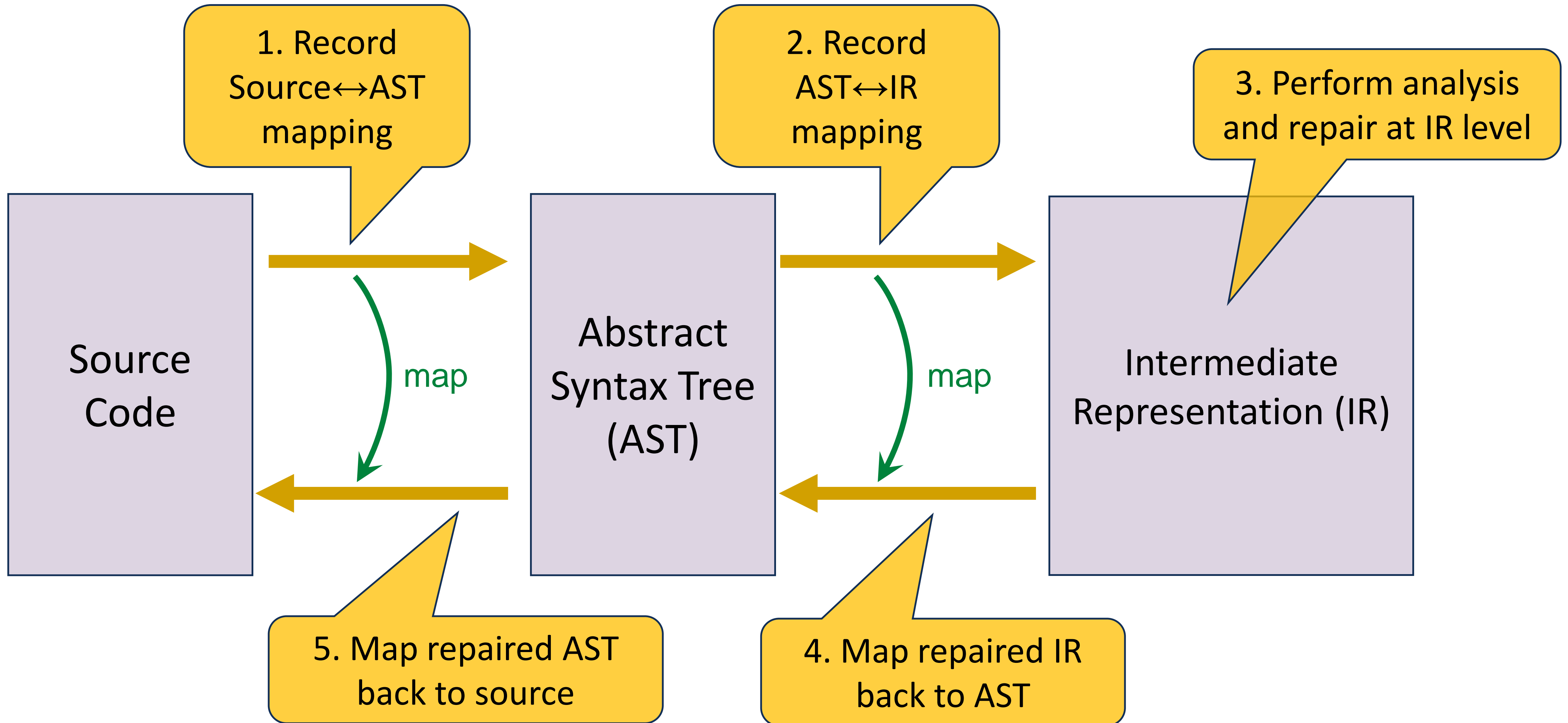
Gap between static analysis and repair of source code



Difficulty: must map repaired IR back to source code.

© 2019 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Source Code Repair Pipeline



© 2019 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Fat pointers

Our chief method of ensuring spatial memory safety is to replace raw pointers with so-called **fat pointers**, which include bound information.

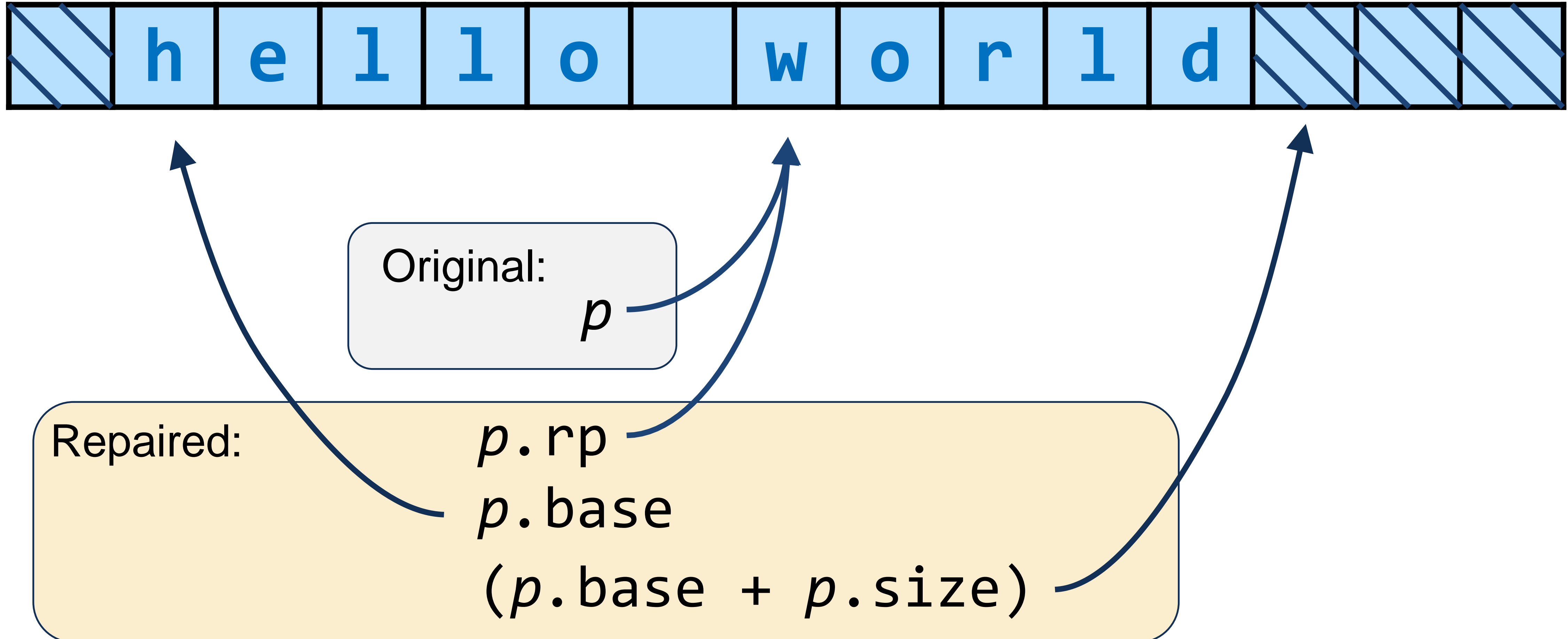
- Before dereferencing a fat pointer, a bounds check is performed.
- For each pointer type *T**, we introduce a new struct definition:

```
struct FatPtr_T {
    T*       rp;    /* raw pointer */
    char*    base;  /* of allocated memory region */
    size_t   size;  /* of allocated memory region, in bytes */
};
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Fat pointer example



© 2019 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Example of repairs to source code

Original Source Code

```
#define BUF_SIZE 256
char nondet_char();

int main() {
    char* p = malloc(BUF_SIZE);
    char c;
    while ((c = nondet_char()) != 0) {
        *p = c;
        p = p + 1;
    }
    return 0;
}
```

Repaired Source Code

```
#include "fat_header.h"
#include "fat_stdlib.h"
#define BUF_SIZE 256
char nondet_char();

int main() {
    FatPtr_char p = fatmalloc_char(BUF_SIZE);
    char c;
    while ((c = nondet_char()) != 0) {
        *bound_check(p) = c;
        p = fatp_add(p, 1);
    }
    return 0;
}
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Auxiliary functions and macros for fat pointers

- `fatmalloc(size)`: Calls `malloc` to allocate memory and returns a fat pointer.
- `fatp_add(fat_ptr, delta)`: Pointer arithmetic (keeping `base` and `size` fields unchanged).
- `bound_check(fat_ptr)`: Aborts if `fat_ptr` is out-of-bounds; else, returns the raw pointer.
- `cast_fatptr(to_type, expr, from_type)`: Type cast between fat types.
- `fat_addr_of_var(var, fat_type)`: Returns a fat pointer to the variable.
- `field_addr_`*T*`(fp, `*F*`)`: Given a fat pointer `fp` to a struct *T* with field *F*; returns a fat pointer to `fp.rp->`*F*.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Wrapper for memory allocation function

For each pointer type $T*$, we define a wrapper around `malloc`:

```
static inline FatPtr_T fatmalloc_T(size_t size) {
    FatPtr_T ret;
    ret.rp   = malloc(size);
    ret.base = (char*) ret.ptr;
    ret.size = size;
    if (!ret.ptr) {ret.size=0;}
    return ret;
}
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Fat pointer arithmetic

Defined as a function for each type *T* :

```
static inline FatPtr_T fatp_add_T(FatPtr_T fp, ptrdiff_t i) {
    FatPtr_T ret = fp;
    ret.rp += i;
    return ret;
}
```

Alternatively, defined as a macro with GNU `typeof` and statement-expressions:

```
#define fatp_add(p_expr, i)            \
  ({ typeof(p_expr) _p = (p_expr);    \
     _p.rp += i;                      \
     _p; })
```

Can also be defined using C11 `_Generic` feature

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Fat pointer bounds checks

Defined as a function, for each type *T*:

```
static inline T bound_check_T(FatPtr_T fp) {
    if (!(fp.base <= (char*) fp.rp &&
        (char*) fp.rp < fp.base + fp.size)) {abort();}
    return ret.rp;
}
```

Alternatively, defined as a macro with GNU `typeof` and statement-expressions:

```
#define bound_check(p_expr)                                  \
  ({ typeof(p_expr) _p = (p_expr);                           \
     if (!(_p.base <= (char*) _p.rp &&                       \
         (char*) _p.rp < _p.base + _p.size)) {abort();};    \
    _p.rp; })
```

Can also be defined using C11 `_Generic` feature

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Limitations

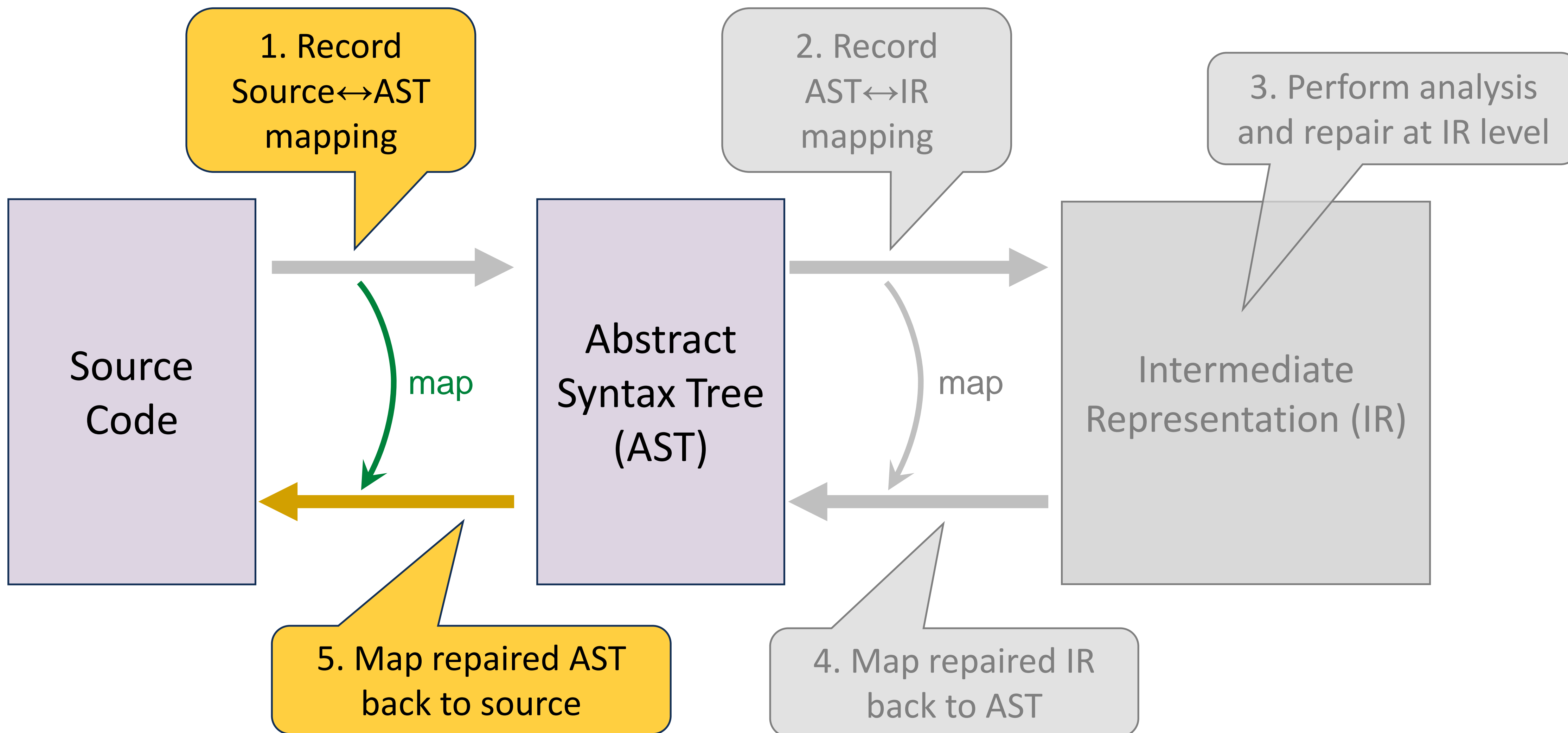
No guarantee of memory safety in presence of

- Concurrency
- External libraries that access program memory
  - but: we are investigating techniques to analyze and repair x86 binary libraries
- Non-standard pointer tricks (e.g., XOR-linked lists)
- Reuse of memory for different types (except via `unions`)
- Dynamically loaded code (including JIT)
- Anything else that interacts poorly with fat pointers

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Source Code Repair Pipeline



© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Abstract syntax tree (AST) ↔ source code

- We implemented of a modification to Clang to extract a Source ↔ AST mapping.
- In translating repairs from AST to source, the C preprocessor is main difficulty.
  - Repairs to macro uses
  - Repairs to `#included` code
  - Conditional-compilation directives (`#ifdef`, `#endif`, etc.) inside expressions
- Considerations of whitespace (e.g., indenting inserted code)
- When an expression or statement is repaired:
  - We generate new source code from the AST.
  - But if a child AST node is unchanged, we re-use its existing source code.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Multiple build configurations

The C preprocessor can conditionally include or exclude pieces of C code depending on the configuration chosen at compile time.

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

Examples of configuration options:

- target platform (e.g., Windows, Linux)
- including or excluding certain features (e.g., FIPS-compliant mode in OpenSSL)
- debug vs. release mode

The final repaired source code should be correct under all desired configurations.

We repair configurations separately and then merge the results.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Enumerating build configurations

We want to find a set of configurations that, together, cover every possible line in the source code text.

For every `#if` directive, we want a configuration under which it evaluates to false.

We will use the Vampyr/Undertaker tools [1] to perform this task.

[1] Reinhard Tartler et al. "Static analysis of variability in system software: The 90,000 `#ifdefs` issue." *USENIX Annual Technical Conference,* 2014.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

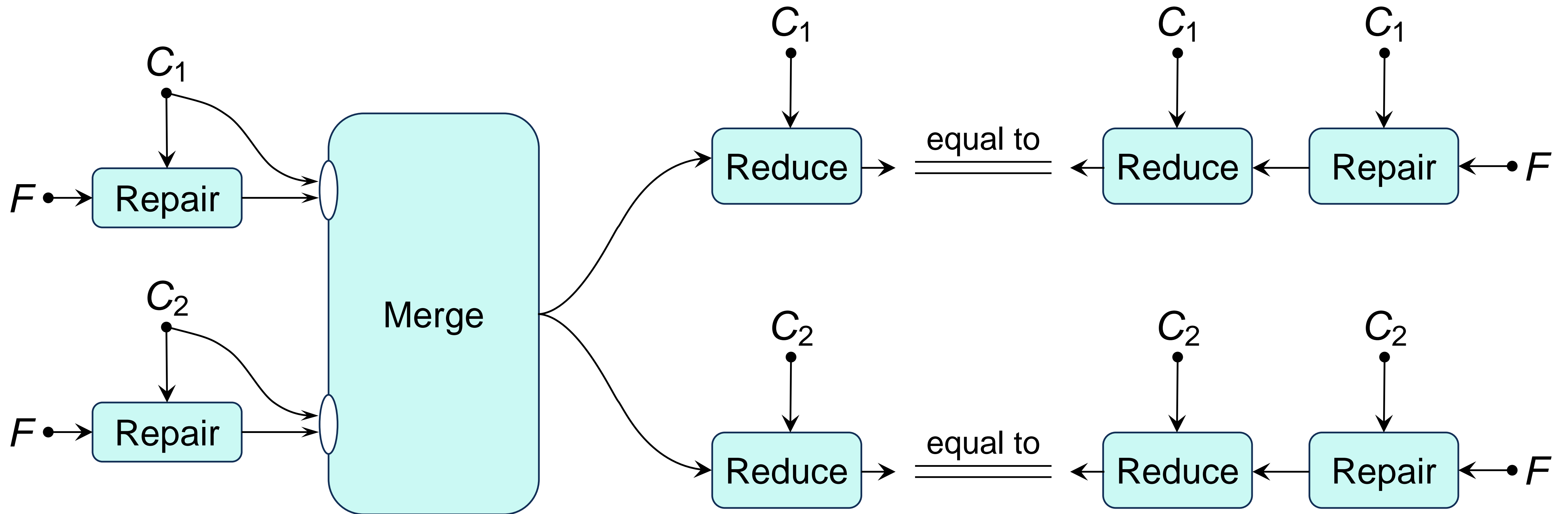
Repairing multiple build configurations

Algorithm:

1. Repair each configuration individually.
2. Merge all the repairs at the level of (unpreprocessed) source code, inserting additional `#if` directives for lines of code that are repaired differently for different configurations, so that:
   - For each configuration *C*, the merged file, when reduced under *C*, is identical (at the token level) to the result of reducing the result of the repairing the original file under configuration *C*.
     - By "reduced by *C*", we mean the result of preprocessing only the conditional-compilation directives under *C*.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.



The Reduce function preprocesses only the conditional-compilation directives.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Example merge of build configurations

Original:

```
foo(
#ifdef LONG
 long* x
#else
 int* x
#endif
)
```

Repaired Config 1:

```
foo(
#ifdef LONG
 FatPtr_long x
#else
 int* x
#endif
)
```

Repaired Config 2:

```
foo(
#ifdef LONG
 long* x
#else
 FatPtr_int x
#endif
)
```

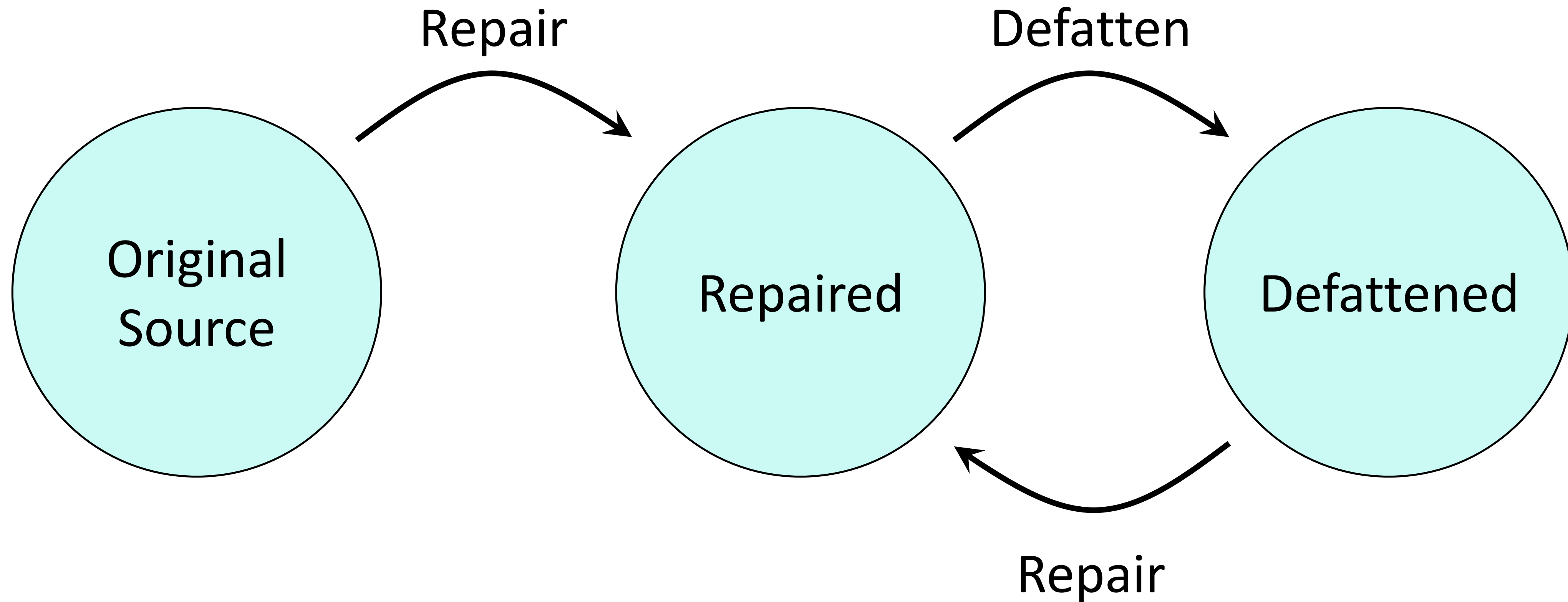
Merged:

```
foo(
#ifdef LONG
 FatPtr_long x
#else
 FatPtr_int x
#endif
)
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Idempotence and Defattening



© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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, the slowdown was 436% (Keaton 2014).
- CETS (ISMM 2010): Compiler-Enforced Temporal Safety associates a unique label with each allocated object. It 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. It cannot prove 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. It requires 64-bit pointers and memory overcommit, and works on Linux.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Conclusion, Status Summary, and Future Work

CURRENT	FY 2020	FUTURE
• Our tool repairs codebases by using fat pointers to ensure memory safety. • Finishing year two of a three-year project • Works on small test cases • Fixing remaining bugs and adding features to handle the SPEC2006 benchmarks	• Optimize and remove unnecessary bounds checks and fattenings • Option to store bounds metadata in side tables instead of modifying memory layout • Work with DoD transition partners to evaluate our tool for DoD use.	• Extend to C++ and other types of repair. • Increase level of automation. • We are happy to work with additional interested DoD transition partners.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Backup Slides

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Related Work

Delete binary column

	Analyze Source	Repair Source	Analyze Binary	Ensure Mem Safety	Whole-program analysis to minimize overhead
LSI 09	Yes	Yes	Yes	S+T	Yes
Softbound	Yes	No	No	S	No
CETS	Yes	No	No	T	No
DangNull	Yes	No	No	T *	No
STONESOUP	Yes	No	Yes	No	N/A
OpenRefactory	Yes	Yes	No	No	N/A

*DangNull zeroes out pointers only on the heap.

S: Spatial
T: Temporal

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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);
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Syntactically ill-formed fragments inside #ifdefs

Repairing the true branch may break the false branch:

```
#ifdef RE_ENABLE_I18N
        if (dfa->mb_cur_max > 1)
            bitset_merge (accepts, dfa->sb_char);
        else {
#endif
            /* (new statement inserted here) */
            bitset_set_all (accepts);
        }
```

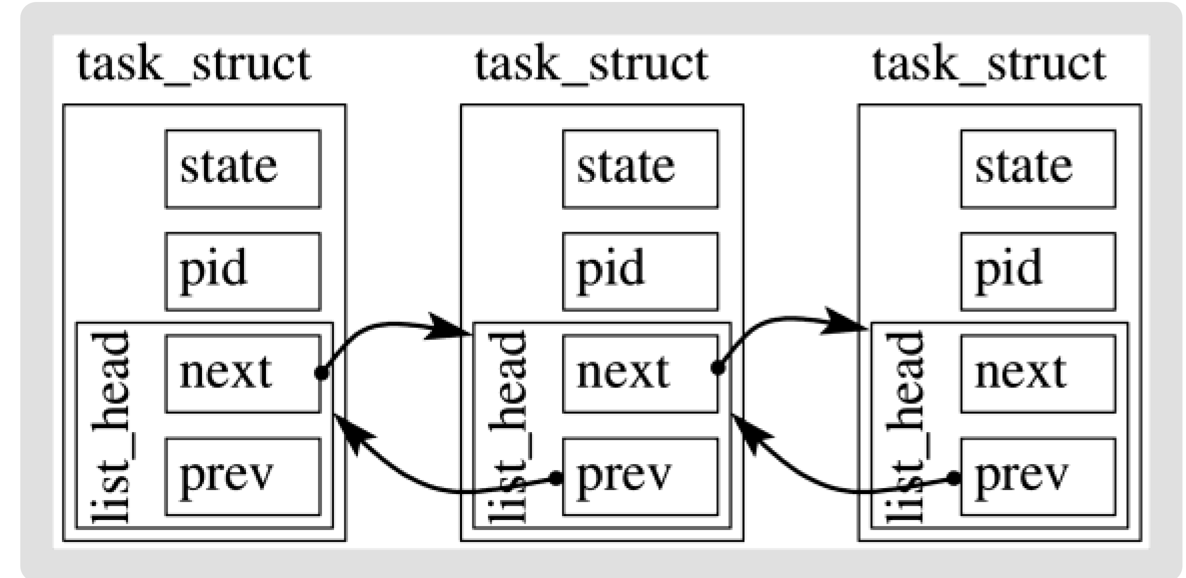

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Example 2: 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 container_of(ptr, type, member) \
  ((type *)((char *)(ptr) - offsetof(type,member)))

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


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Original Source Code

```
1
2   #define container_of(ptr, type, member) \
3     ((type *)((char *)(ptr) - offsetof(type, member)))
4
5   struct list_head {
6       struct list_head *next, *prev;
7   };
8
9   struct node {
10      int              value;
11      struct list_head  linkage;
12      struct list_head  nested;
13  };
14
15
16  static void inspect(const struct list_head *head) {
17      assert(head != 0);
18      assert(head->next != head);
19      struct node *node =
20          container_of(head, struct node, linkage);
21
22      assert(node->nested.next == &node->nested);
23  }
24
```

Repaired

```
#include "fat_header.h"
Declare_FatPtr_Type(struct list_head, FatPtr_list_head, list_head);
Declare_FatPtr_Type(struct node, FatPtr_node, node);
#include "fat_stdlib.h"
struct list_head {
    struct FatPtr_list_head next;
    struct FatPtr_list_head prev;
};
struct node {
    int value;
    struct list_head linkage;
    struct list_head nested;
};
define_struct_field_address(node, nested, FatPtr_list_head);
#define field_addr_node(fp, field) sfa_node__ ## field(fp)
static void inspect(struct FatPtr_list_head head) {
    assert(head.rp != 0);
    assert((*bound_check(head)).next.rp != head.rp);
    struct FatPtr_node node = cast_fatptr(FatPtr_node,
        fatp_add(cast_fatptr(FatPtr_char, head, FatPtr_list_head),
                -offsetof(struct node, linkage)), FatPtr_char);
    assert((*bound_check(node)).nested.next.rp ==
        field_addr_node(node, nested).rp);
}
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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:;
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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.rp` and a bound check is prepended:
  `if (!(x.base <= x.rp && x.rp < x.base + x.size)) {abort();}`
- When passed to a standard-library function, *x* is changed to `x.rp` and a bound check is inserted before.
- Subtraction of pointers `(p - q)` is changed to subtraction of the `rp` fields of fat pointers `(p.rp - q.rp)`. Comparison operators are handled the same way. Addition and subtraction 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.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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 currently allocated region returned by `malloc(·)`.
- Possible executions include those where **gaps of unallocated memory** are left between local variables on a stack frame. (Padding in `structs` **is not** a gap of unallocated memory.)
- A heap region gets allocated when returned by `malloc(·)`, and gets deallocated when `free`'d.
- A region in a function's stack frame become allocated on entry to the function.
  It becomes deallocated on exit from the function (including via `longjmp`, etc.).
- A **proof of memory safety** is often divided into two parts:
  1. **Spatial:** Writing or reading beyond the bounds of a memory region.
  2. **Temporal:** Writing or reading to a region after it has been deallocated.
- 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.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Measurements and Goals

Runtime overhead for spatial repairs: less than 10% of base runtime.

Runtime overhead for temporal repairs: less than 15% of base runtime.

Code bloat: less than 15% for typical programs (by source-code and binary file sizes)

Benchmarks: SPEC CPU2006 (e.g., bzip2, gcc, Markov model, speech recognition), PapaBench (embedded real-time software for UAV), svcomp, Jasper, and others.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Fat pointer type casts

Functions to casting to and from `FatPtr_void`, for each type *T*:

```
static inline FatPtr_void cast_T_to_fatvoid(FatPtr_T fatp) {
    FatPtr_void ret = {(void*)fatp.rp, fatp.base, fatp.size};
    return ret;
}
static inline FatPtr_T cast_fatvoid_to_T(FatPtr_void fatvp) {
    FatPtr_T ret = {(T*)fatvp.rp, fatvp.base, fatvp.size};
    return ret;
}
```

Macro for casts between arbitrary fat-pointer types:

```
#define cast_fatptr(to_type, expr, from_type)  \
    cast_fatvoid_to_ ## to_type(cast_ ## from_type ##
_to_fatvoid(expr))
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Fat-pointer address-of-field-of-struct

Define as a macro:

```
#define define_struct_field_access(struct_type, field, field_fat_type)  \\
    static inline field_fat_type sfa_ ## struct_type ## __ ## field (   \\
                                        FatPtr_ ## struct_type s) { \\
        field_fat_type ret = {&s.rp->field, s.base, s.size};           \\
        return ret;                                                     \\
    }
```

This macro is used once for each field of each struct.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Example of wrapper defined in `fat_stdlib.h` header

```
static inline FatPtr_void fat_memcpy(FatPtr_void dest,
                                     FatPtr_void src,
                                     size_t      n)
{
    if ((char*)dest.rp < dest.base) {abort();}
    if ((char*)src.rp  < src.base)  {abort();}
    if (((char*)dest.rp - dest.base) + n >= dest.size) {abort();}
    if (((char*)src.rp  - src.base ) + n >= src.size)  {abort();}
    memcpy(dest.rp, src.rp, n);
    return dest;
}
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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.

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Normalization of `structs` and `unions`

- In C, `structs` and `unions` can be defined within a function definition.
- However, our repair relies on introducing auxiliary functions.
- So, we move the `struct`/`union` definition to file scope and add a `typedef` if unnamed.

Example original code:

```
float foo(struct {float x; float y;} *point) {
    return point->x * point->x + point->y * point->y;
}
```

Normalized:

```
typedef struct {float x; float y;} anon_recd_dd59b0;
float foo(anon_recd_dd59b0 *point) {
    return point->x * point->x + point->y * point->y;
}
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Leakage of sensitive information in re-used buffer

Buffer contents after **first HTTP request**:

"	p	a	s	s	w	o	r	d	"	:	"	h	u	n	t	e	r	2	"

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

"	s	o	r	t	"	:	"	i	d	"	}	h	u	n	t	e	r	2	"

Upper bound for reading: most recently written location

© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

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...");
```


© 2019 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.