RESEARCH REVIEW 2019

Automated Code Repair (ACR) to Ensure Memory Safety

Dr. Will Klieber (presenting)
Ryan Steele
Matt Churilla
Derek Leung
David Svoboda
Mike McCall
Ruben Martins (CMU SCS)

[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.

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

DM19-1104

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, according to https://techcrunch.com/2019/08/31/china-google-iphone-uyghur/)
  - 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.

Automated Code Repair (ACR) for Memory Safety

Solution: Automatically repair source code to assure memory safety.

- Abort program before memory violation.

Approach:

- Transform source code to an intermediate representation (IR), retaining mapping.
- Try to assure that each memory access:
  - is within bounds (spatial memory safety), and
  - is not to a deallocated region (temporal memory safety — future work for FY20)
- If unable to assure, repair code to ensure memory safety.
  - Use ***fat pointers*** to store bounds information where possible.
- Map the repairs at the IR level 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 support C and plan to have limited support for C++

Currently finishing year 2 of a 3-year project.

ACR Tool

Envisioned use of tool

- Use before every release build
- Use occasionally for debugging builds
- Intended for ordinary developers
- Can be a tool in the DevOps toolchain
- Can be used for legacy code and for new code

© 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.

A small runtime overhead is often acceptable.

© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

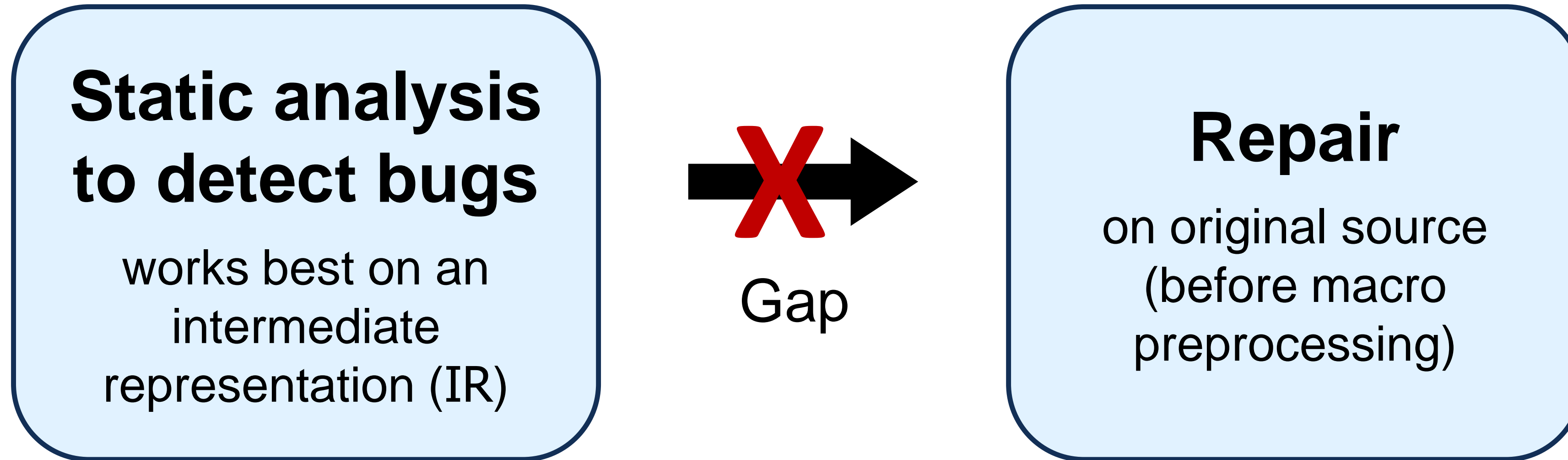
Why repair of source code instead of as a compiler pass?

Repair of source code	Repair as a compiler pass
Easily audited (if desired).	Must trust the tool.
Repairs can easily be tweaked to improve performance, if necessary.	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.
Okay to do slow, heavy-weight static analysis; produces a persistent artifact.	Slowing down every test build is not okay.

© 2019 Carnegie Mellon University

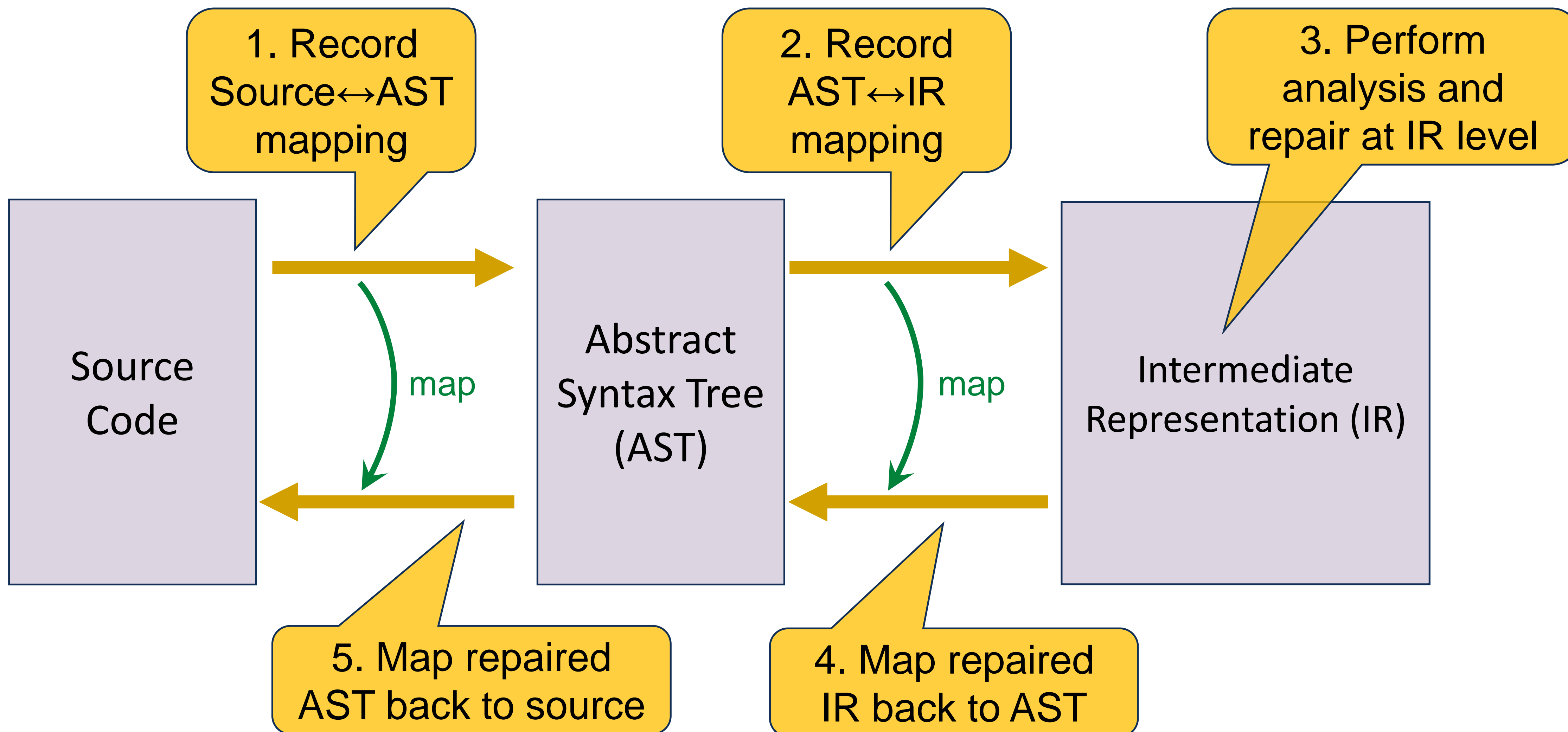
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Gap between static analysis and repair of 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

We replace raw pointers with **fat pointers**:

- A *fat pointer* is a `struct` that includes the pointer itself as well as bounds information.
- Before dereferencing a fat pointer, a bounds check is performed.
- For each pointer type $T*$, we introduce a fat-pointer type defined as follows:

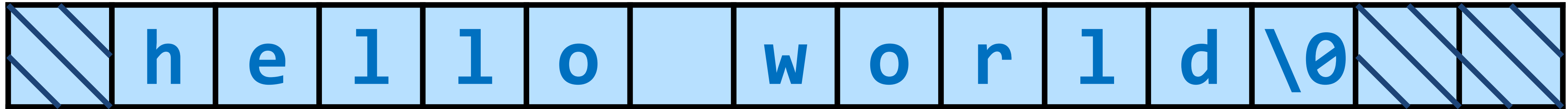
```
struct FatPtr_T {
    T*       rp;    /* raw pointer */
    char*    base;  /* of allocated memory region */
    size_t   size;  /* of allocated memory region, in bytes */
};
```

Fattening of pointers has been performed as a compiler pass:

- Todd Austin et al. “Efficient detection of all pointer and array access errors.” *PLDI,* 1994.
- Wei Xu et al. “An efficient and backwards-compatible transformation to ensure memory safety of C programs.” *ACM SIGSOFT,* 2004.

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

Fat pointer example



Original:

p

Repaired:

p.rp

p.base

(p.base + p.size - 1)

© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Example of tool output

Original Source Code

```
1
2
3  #define BUF_SIZE 256
4  char nondet_char();
5
6  int main() {
7      char* p = malloc(BUF_SIZE);
8      char c;
9      while ((c = nondet_char()) != 0) {
10         *p = c;
11         p = p + 1;
12     }
13     return 0;
14 }
```

Repaired Source Code

```
1  #include "fat_header.h"
2  #include "fat_stdlib.h"
3  #define BUF_SIZE 256
4  char nondet_char();
5
6  int main() {
7      FatPtr_char p = fatmalloc_char(BUF_SIZE);
8      char c;
9      while ((c = nondet_char()) != 0) {
10         *bound_check(p) = c;
11         p = fatp_add(p, 1);
12     }
13     return 0;
14 }
```


© 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.rp;
    ret.size = size;
    if (ret.rp == NULL) {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 `typeof` and statement-expressions (for gcc/clang):

```
#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 `typeof` and statement-expressions (for gcc/clang):

```
#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.

External libraries

- Many programs have data structures such as linked lists and trees.
- If such data structures are accessed by external binary code, then the pointers inside them cannot be fattened.
- We identify such pointers using a whole-program points-to analysis with an *allocation-site abstraction*.
- We are also investigating techniques to analyze and repair x86 binary libraries.

© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Limitations

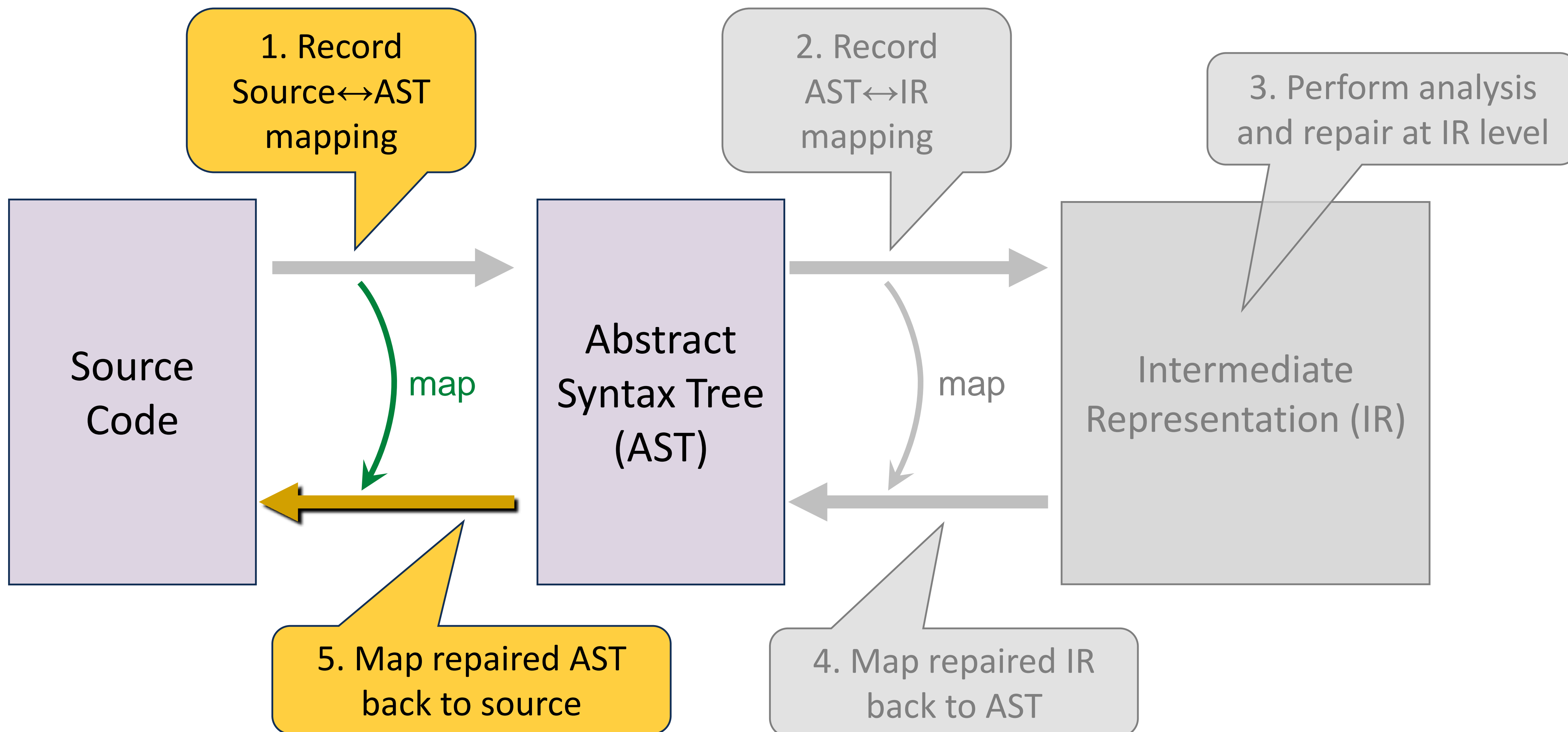
We cannot guarantee memory safety in the presence of:

- External libraries that access program memory
- 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
- Concurrency

© 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 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
- 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 code depending on the configuration chosen at compile time.

- Definition: a *configuration* is a partial assignment of integer values to symbols used in conditional preprocessor directives such as a `#ifdef`.

Examples of options specified in a configuration:

- 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.

Example merge for build configurations

Original:

```
void foo(
#ifdef USE_LONG
  long* x
#else
  int* x
#endif
);
```

Repaired Config 1:

```
void foo(
#ifdef USE_LONG
  FatPtr_long x
#else
  int* x
#endif
);
```

Repaired Config 2:

```
void foo(
#ifdef USE_LONG
  long* x
#else
  FatPtr_int x
#endif
);
```

Merged:

```
void foo(
#ifdef USE_LONG
  FatPtr_long x
#else
  FatPtr_int x
#endif
);
```


© 2019 Carnegie Mellon University

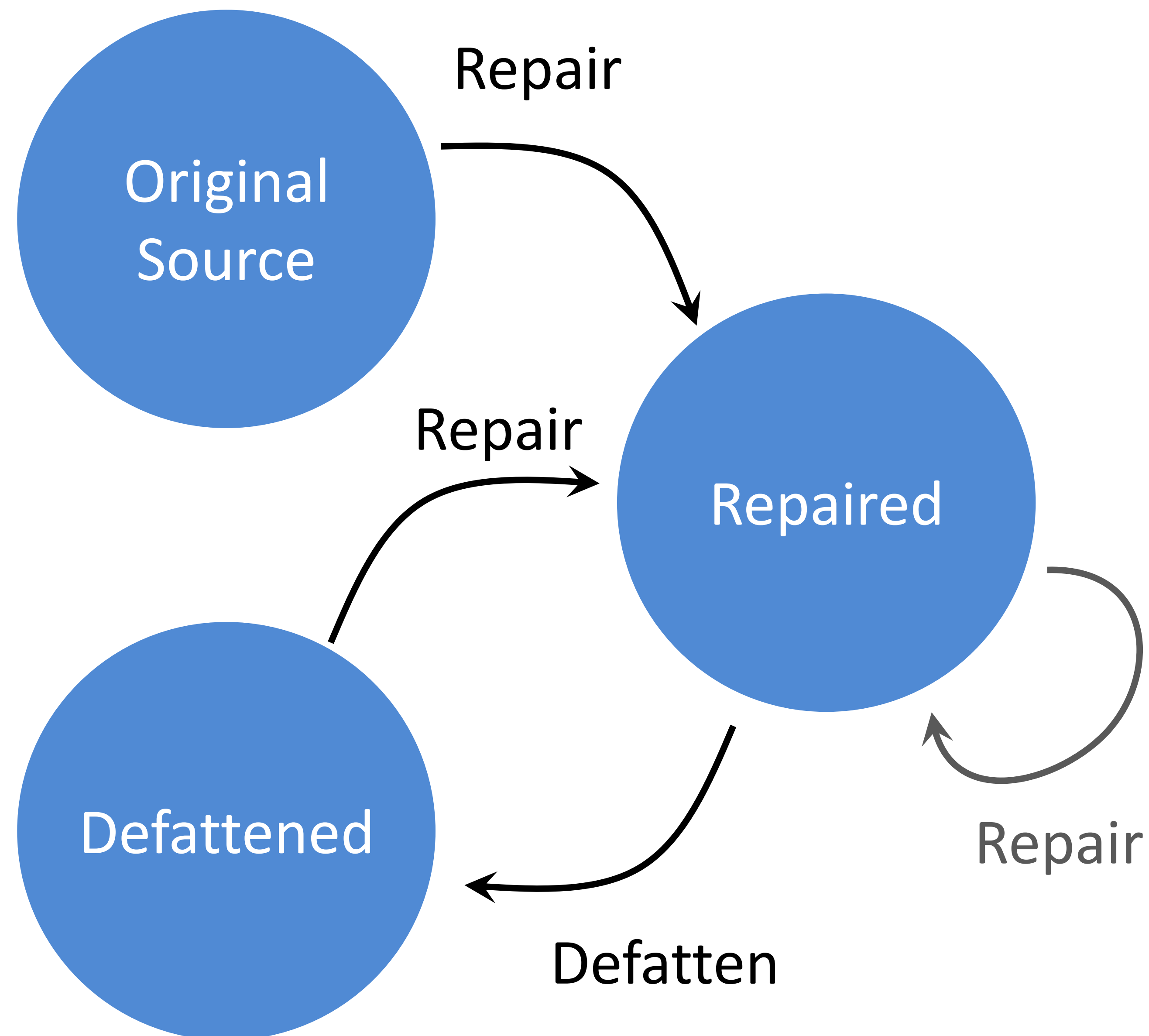
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Idempotence and Defattening

$$\text{repair}(\text{repair}(s)) \stackrel{?}{=} \text{repair}(s)$$

Not yet.

But: (repair ∘ defatten) is idempotent.



© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Project Team

Will Klieber, Ryan Steele, Matt Churilla, Derek Leung
David Svoboda, Mike McCall, Ruben Martins (CMU SCS)

© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Conclusion, Status Summary, and Future Work

Near-term	Mid-term (3-5 years)	Long-term (5-10 years)
• Adding features and fixing remaining bugs to handle the SPEC2006 benchmarks • Optimize to remove unnecessary bounds checks and fattenings • Option to store bounds info in shadow memory • Temporal memory safety • Work with DoD transition partners to evaluate our tool for DoD use	• Extend to other types of repair • Increase level of automation – manual performance tweaks – work with existing static analysis tools • DoD programs and contractors are using automated repair to remove spatial memory bugs from legacy code and in development pipeline.	• Tool can be used by DoD to ensure memory safety (spatial and temporal) as part of all software projects with code written in memory-unsafe languages (such as C and C++).

© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.