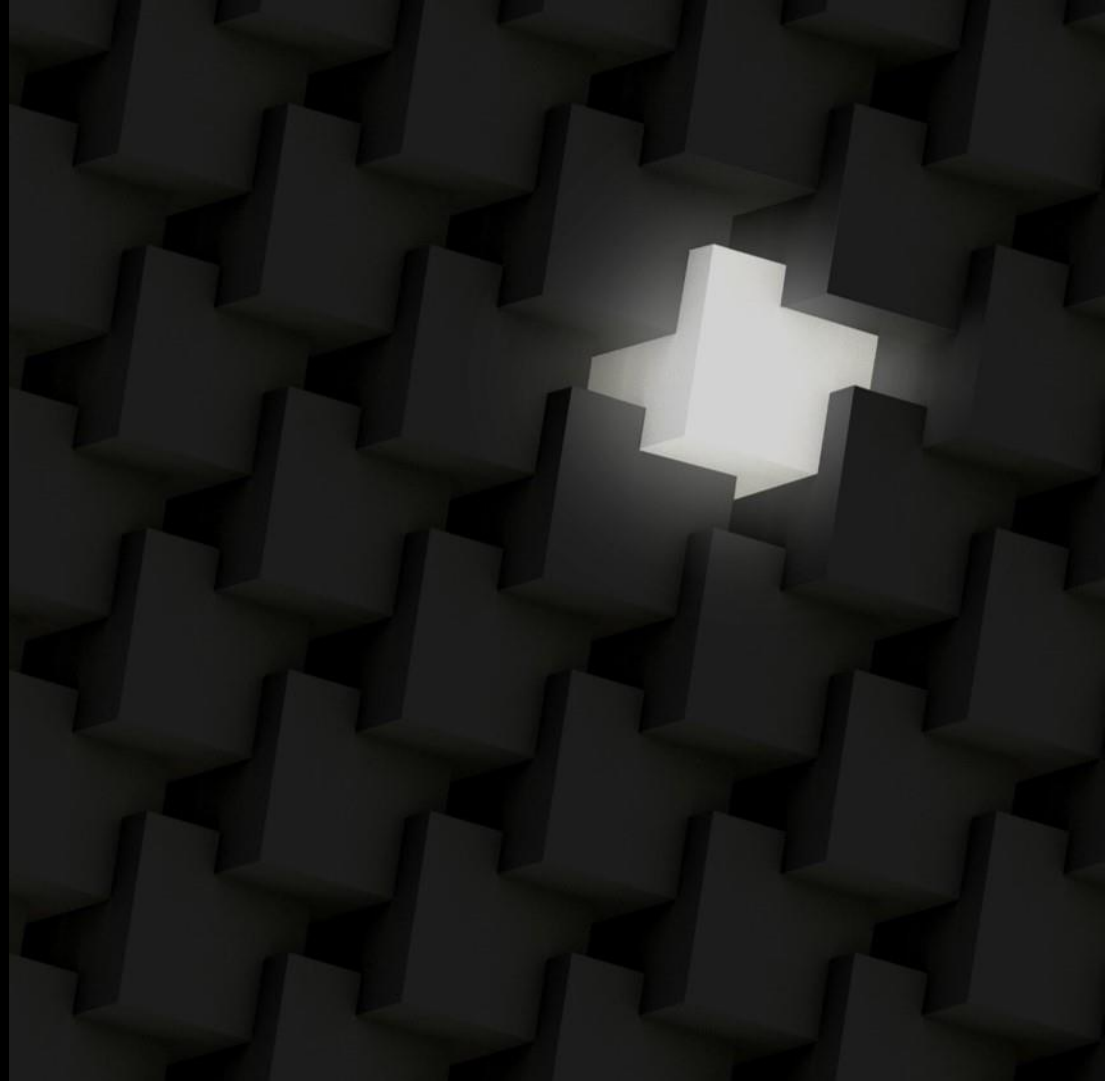
Carnegie Mellon University
Software Engineering Institute

RESEARCH REVIEW 2020

Automated Code Repair (ACR) to Ensure Memory Safety

Will Klieber



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.

DM20-0912

Automated Code Repair (ACR) for Memory Safety

Problem: Software vulnerabilities constitute a major threat to DoD.

- Spatial 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.

Automated Code Repair (ACR) for Memory Safety

Solution: Automatically repair source code to assure spatial memory safety.

- Abort program (or call error-handling routine) before memory violation.

Approach:

- Transform source code to an intermediate representation (IR), retaining mapping.
- Repair program to use ***fat pointers*** to track bounds and insert a bounds check before memory accesses.
- Map the repairs at the IR level back to source code.

Automated Code Repair (ACR) tool as a black box

Input: Buildable codebase written in C

Output: Repaired source code that is still human-readable and maintainable

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 (for new code)
- Can be used for legacy code

Why repair at the source-code level?

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, more error-prone, and create unwanted dependencies.
Okay to do slow, heavy-weight static analysis; produces a persistent artifact.	Slowing down every build is not okay.

Gap between static analysis and repair of source code

Static analysis to detect bugs

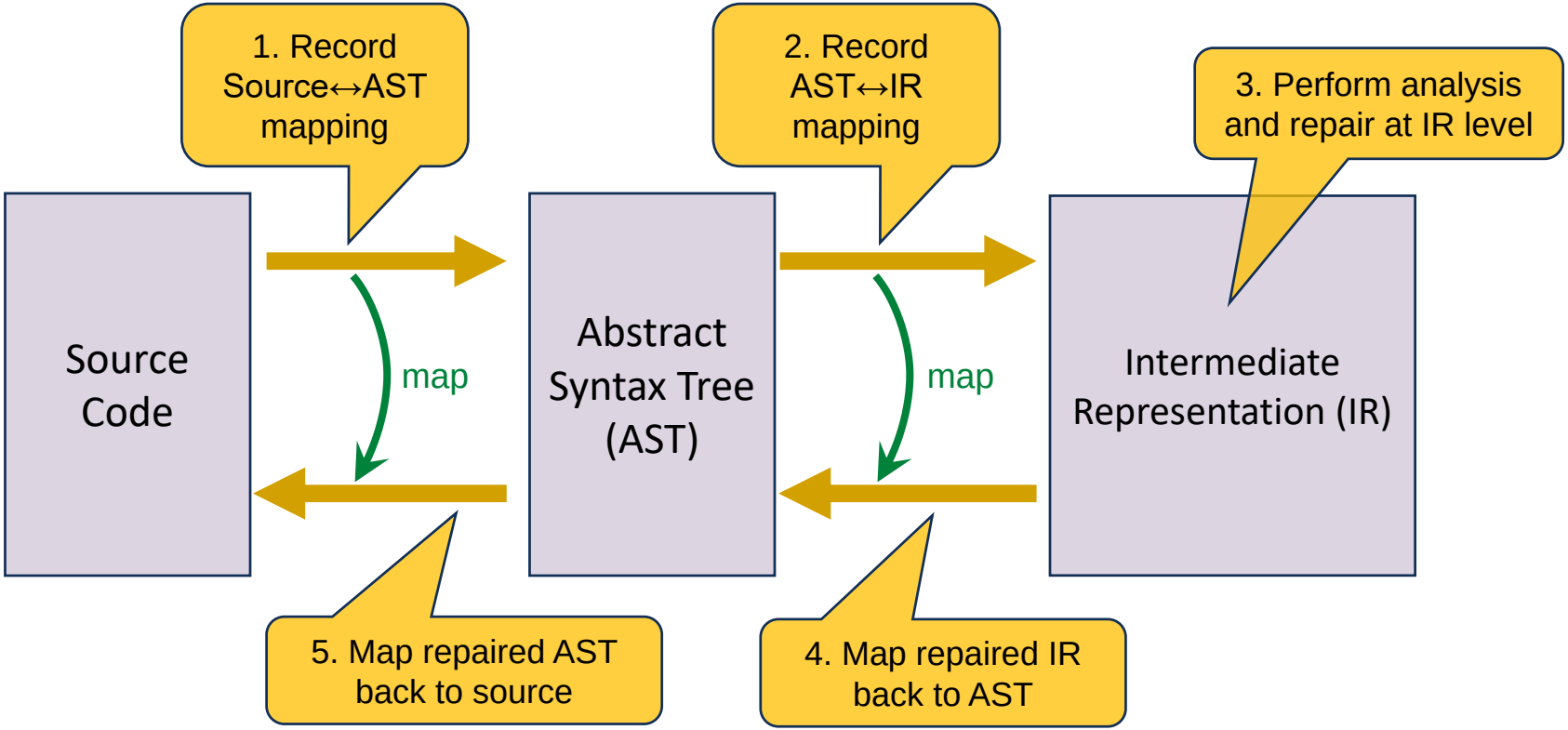
works best on an intermediate representation (IR)

Gap

Repair

on original source (before macro preprocessing)

Source Code Repair Pipeline



Fat pointers

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

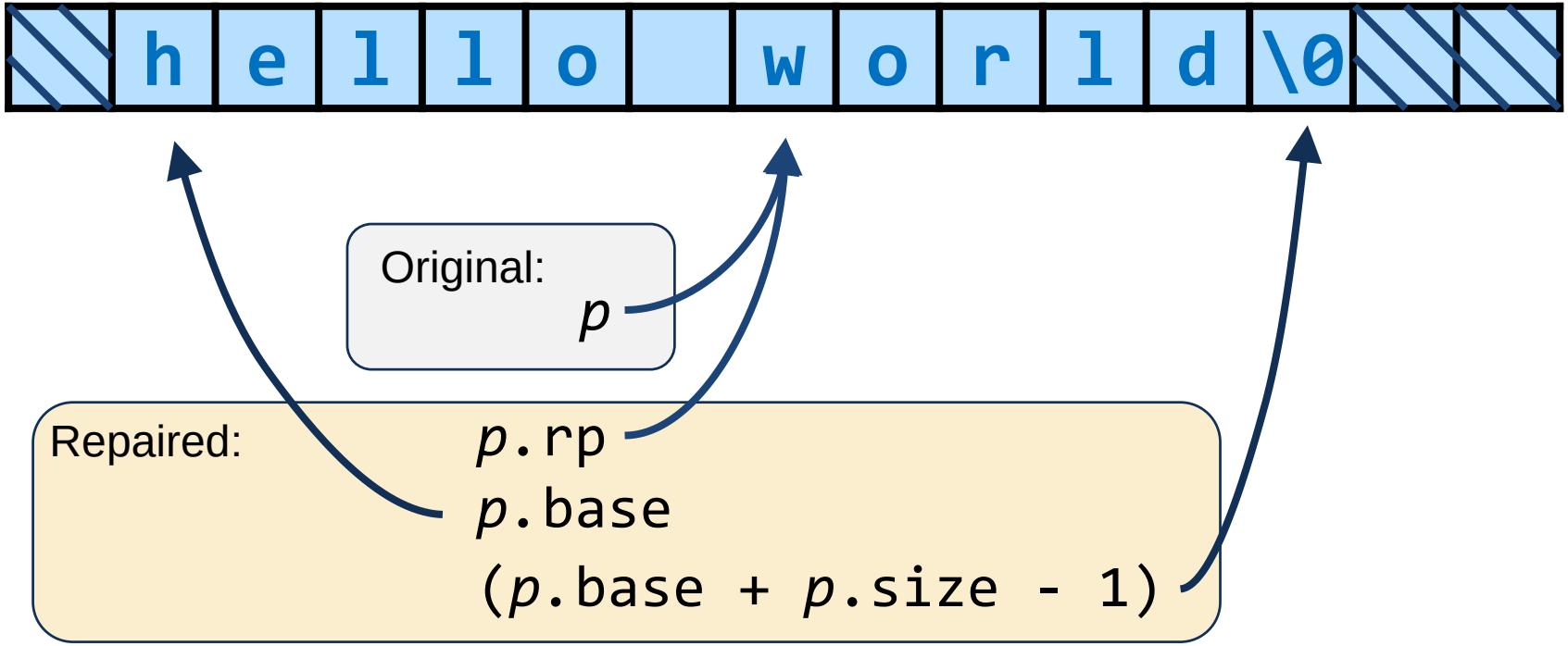
- A *fat pointer* is a `struct` that includes the pointer itself as well as bounds information.
- Before dereferencing a fat pointer, a bounds check is performed.
- For each pointer type $T*$, we introduce a fat-pointer type defined as follows:

```
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.

Fat pointer example



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 }
```

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;
}
```

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 single macro (using widely supported gcc/clang extensions):

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

Can also be defined using C11 `_Generic` feature

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 single macro (using widely supported gcc/clang extensions):

```
#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

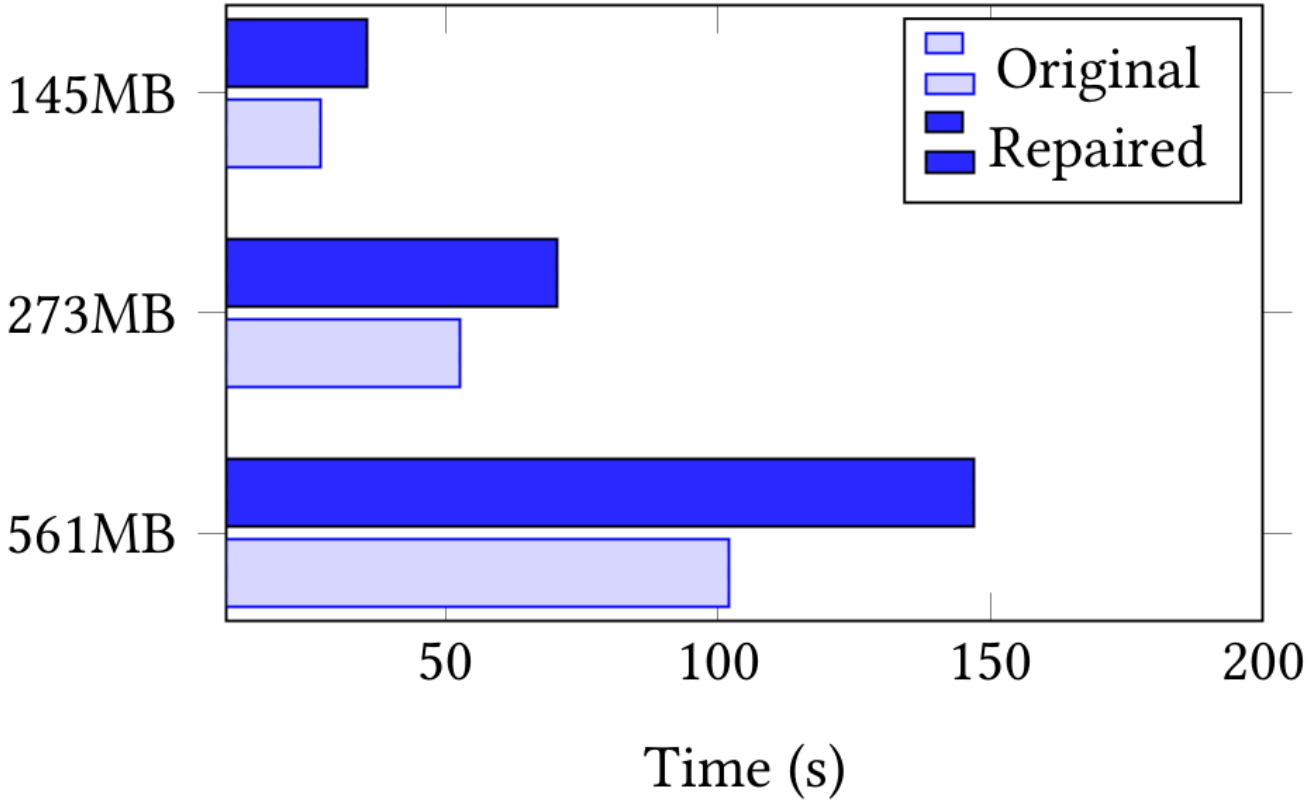
Results – SVCOMP benchmarks

We ran our tool on 52 memory-safety benchmarks in the SVCOMP benchmark suite.

To verify the efficacy of our repairs, we ran Symbiotic (a software-verification tool) on the original and repaired files:

	Safe	Unsafe	Unknown
Original	0	48	4
Repaired	27	0	25

Running time of bzip2 (original vs repaired) on 3 files



Reducing runtime overhead

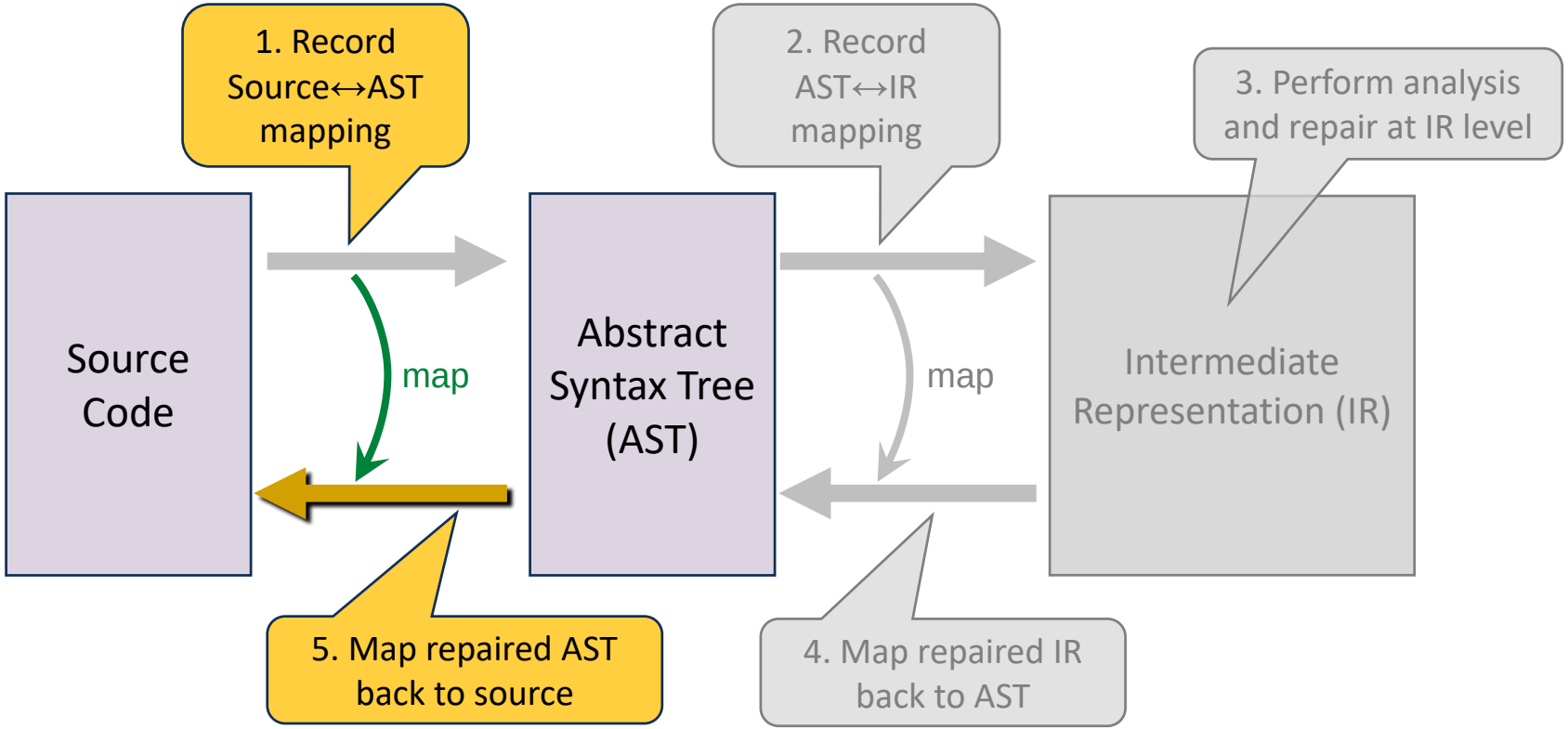
- Our overhead time was around 50% on bzip2.
  - Our DoD collaboration partners said this is too high for many of their use cases.
  - Can we reduce it significantly while still proving memory safety?
    - Probably not, but automated repair is valuable even if it fixes only the likeliest bugs.
- To reduce the overhead time, we added an option to insert bounds checks only for memory accesses that are warned about by an external static analyzer.
  - This **reduced the overhead to 6% on bzip2**.

Limitations

We cannot guarantee memory safety in the presence of:

- Non-standard pointer tricks (e.g., XOR-linked lists)
- Reuse of memory for different types (except via `unions`)
- Concurrency
  - Race conditions can cause memory corruption
- External code that accesses program memory
  - If the program's data structures are accessed by external binary code, the pointers inside them cannot be fattened.
  - We identify such pointers using a whole-program points-to analysis with an *allocation-site* abstraction.

Source Code Repair Pipeline



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
- When an expression or statement is repaired:
  - We generate new source code for the repaired portion of the AST.
  - For unchanged portions of the AST, we re-use the existing source code.

Multiple build configurations

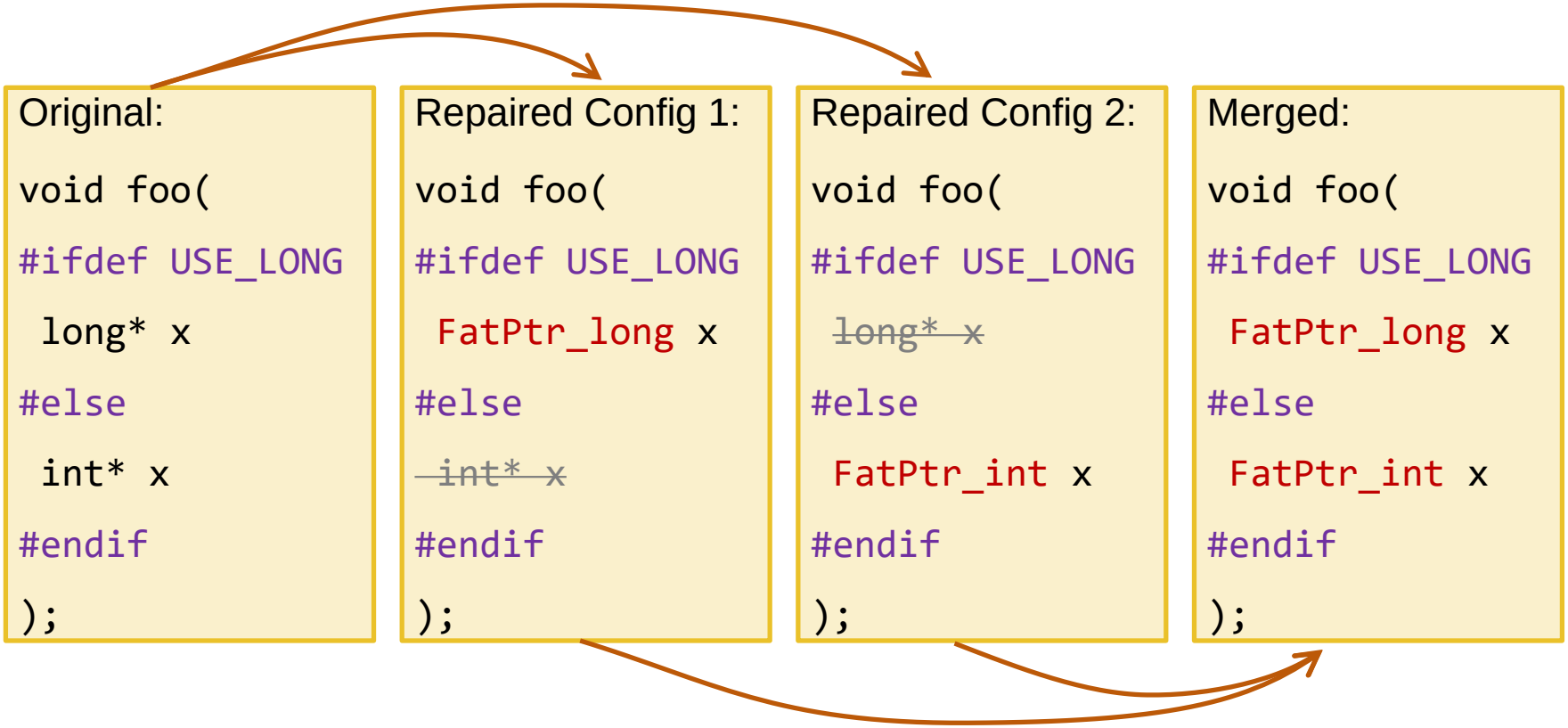
The C preprocessor can conditionally include or exclude pieces of code depending on the configuration chosen at compile time.

- By “configuration”, we mean the values assigned to the symbols used in preprocessor directives such as `#ifdef`.

We repair configurations separately and then merge the results such that the final repaired code is correct under all desired configurations.

- If a line of code is repaired differently for different configurations, then each version is included, guarded by appropriate conditional directives.

Example merge for build configurations

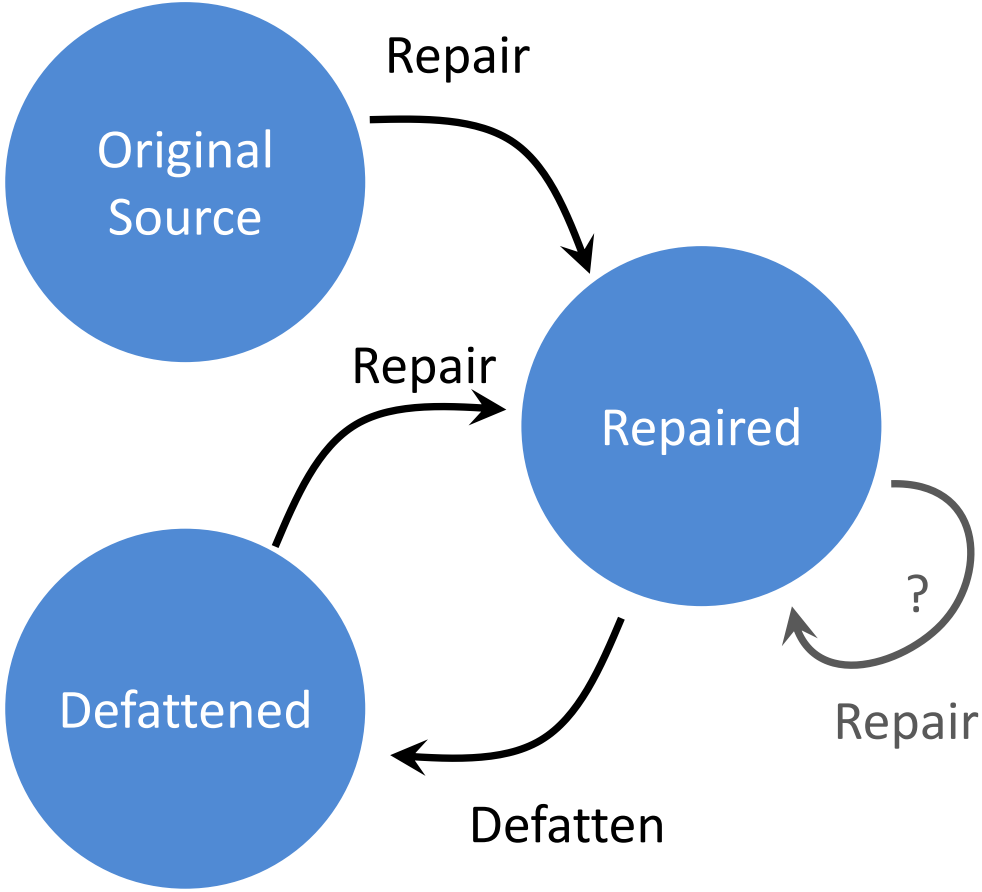


Idempotence and Defattening

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

Not yet.

But: (repair ∘ defatten) is idempotent.

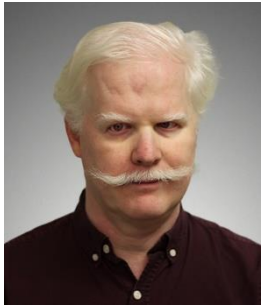


Project Team

Will Klieber

Ryan Steele

Matt Churilla



David Svoboda

Mike McCall

Ruben Martins (CMU SCS)

Conclusion

Our tool repairs codebases to ensure memory safety using fat pointers.

User can select one of two modes:

1. Make all possible automatic repairs (larger runtime overhead, e.g., 50%)
2. Only repair likely memory violations (significantly less runtime overhead)

We are happy to provide our tool to any interested parties.

- This is a research-grade tool, not a production-grade tool.
- Further development work to suit particular needs is possible.

In the long term and with further development, DoD can use this technology to ensure memory safety as part of all software projects with code written in memory-unsafe languages (such as C and C++).

Contact Will Klieber via **info@sei.cmu.edu** or **https://www.sei.cmu.edu/contact-us/**

Blog post: https://insights.sei.cmu.edu/sei_blog/2020/02/automated-code-repair-to-ensure-memory-safety.html