Automated Code Repair to Ensure Memory Safety

Will Klieber
Ryan Steele
Matt Churilla

David Svoboda
Mike McCall
Ruben Martins (CMU)

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.

DM20-0583

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.

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 spatial 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

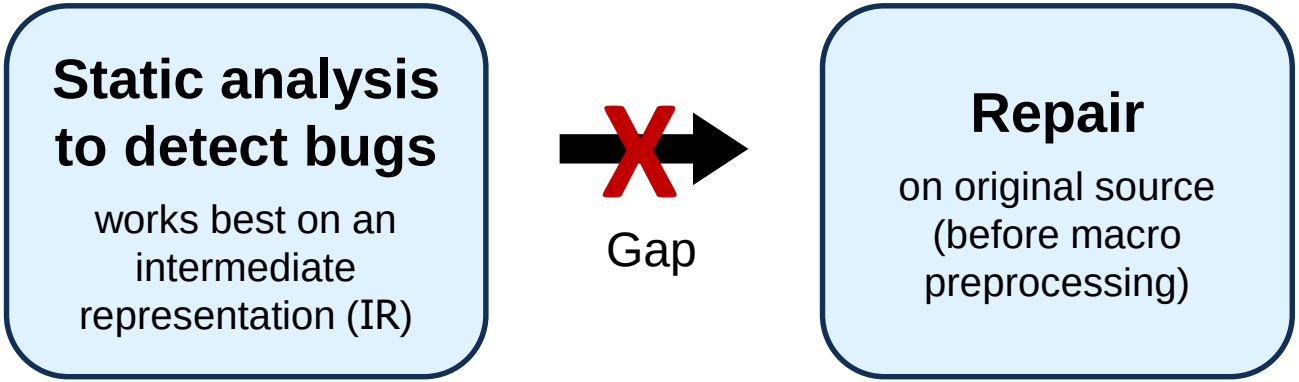
Output: Repaired source code that is still human-readable and maintainable.

We currently support C code. Support for C++ can likely be added without too much difficulty.

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

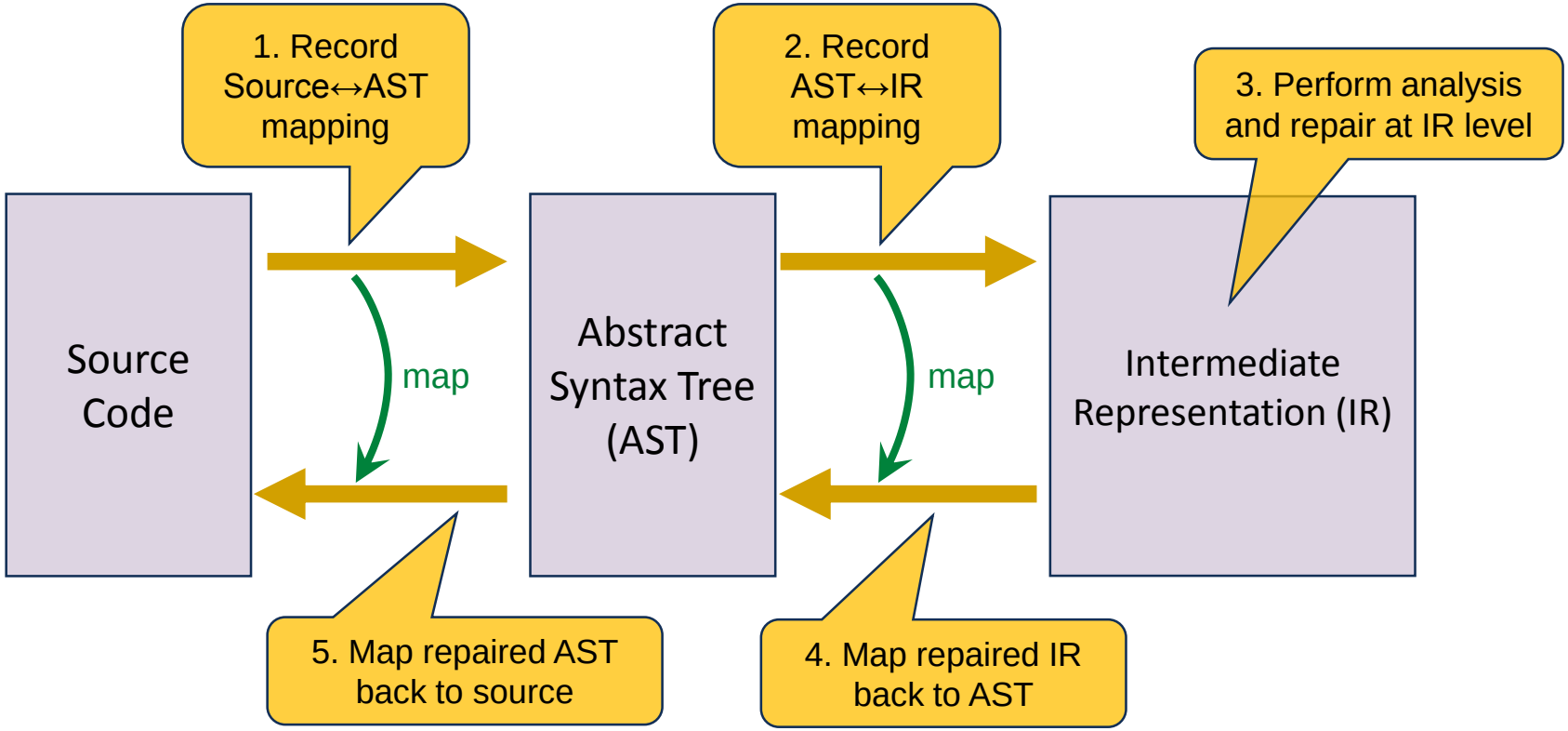
Repair of source code	Repair as a compiler pass
Easily audited (if desired).	Must trust the tool.
Repairs can easily be tweaked to improve performance, if necessary.	Difficult to remediate performance issues caused by repair.
Changes to source code are frequent and easily handled.	Changes to the build 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.

Gap between static analysis and repair of source code



Difficulty: must translate repaired IR back to source code.

Source Code Repair Pipeline



Fat pointers

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

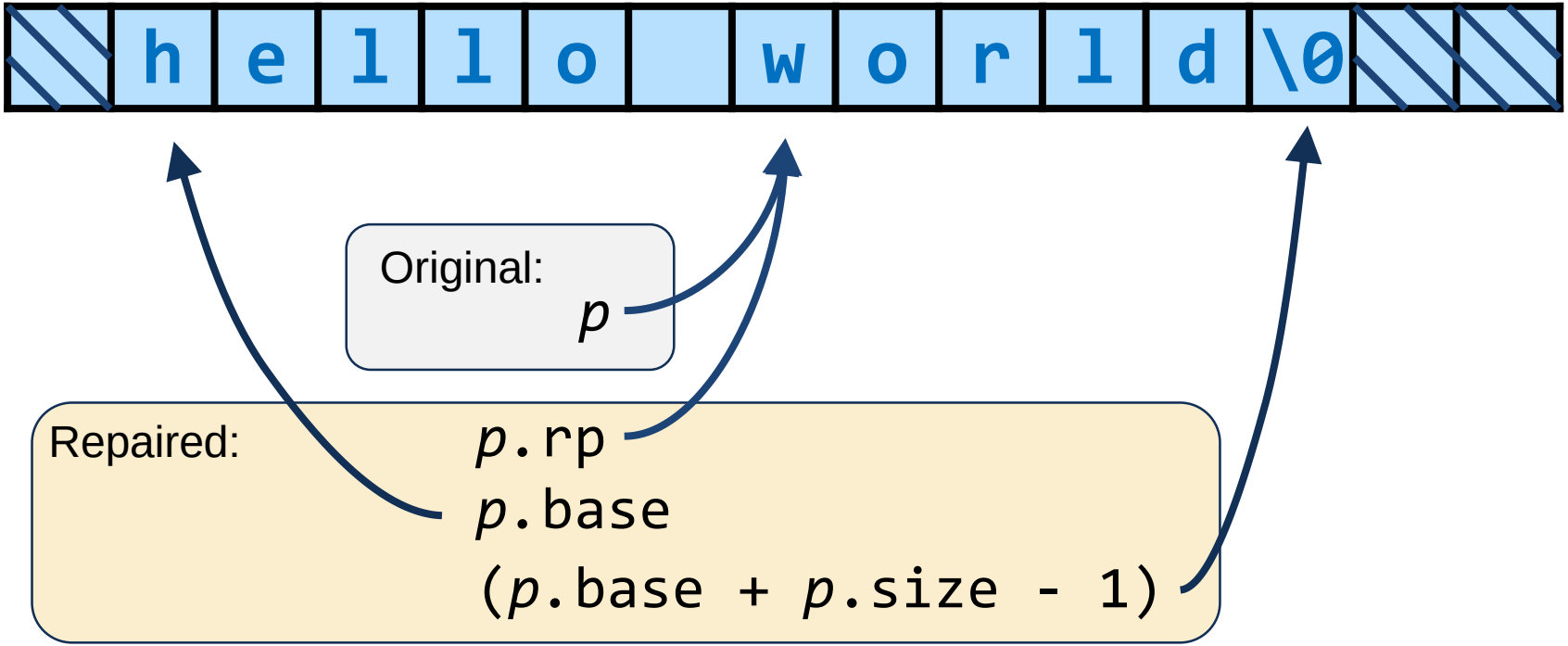
- A *fat pointer* is a `struct` that includes the pointer itself as well as bounds information.
- Before dereferencing a fat pointer, a bounds check is performed.
- For each pointer type $T*$, we introduce a fat-pointer type defined as follows:

```
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 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

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

External libraries

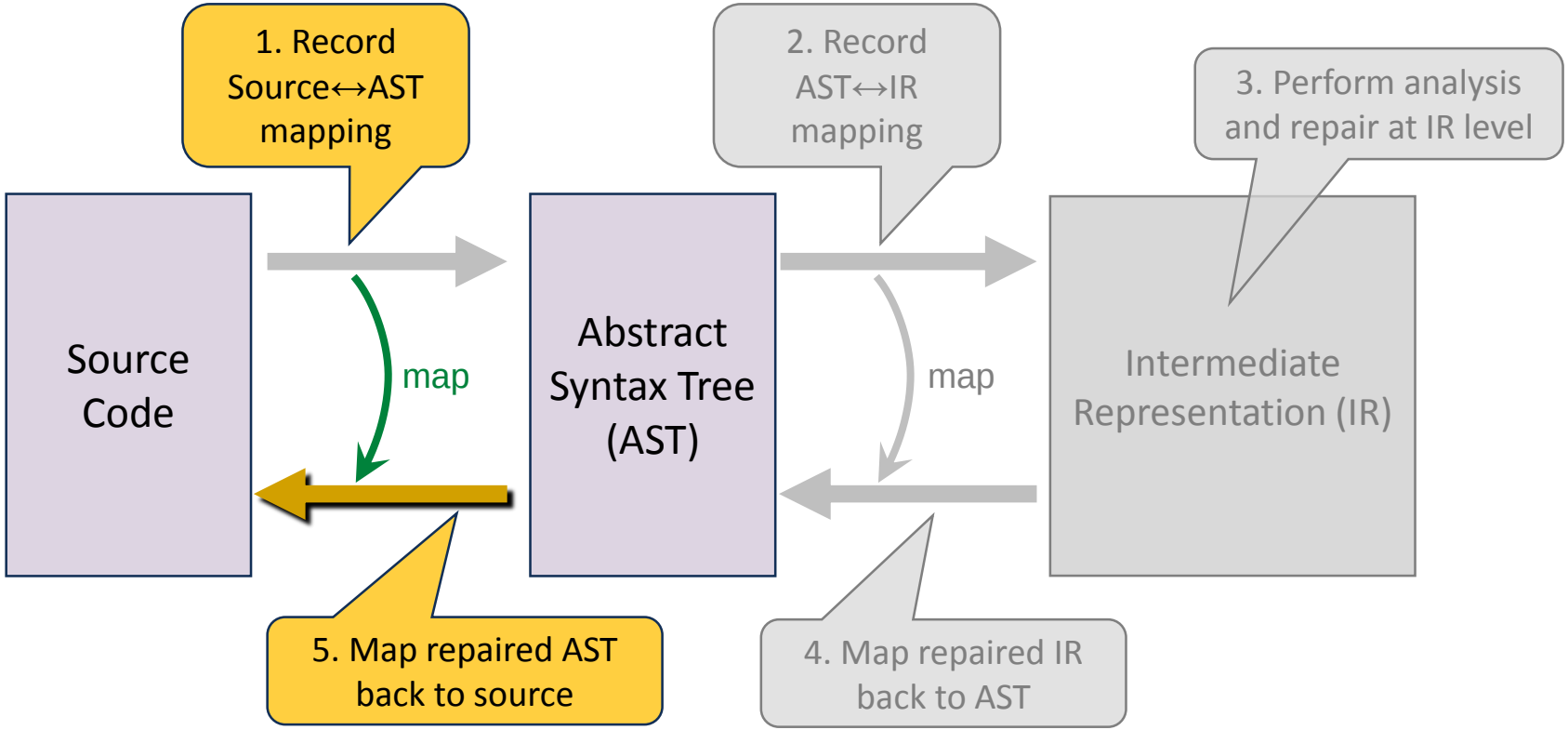
- Many programs have data structures such as linked lists and trees.
- If such data structures are accessed by external binary code, then the pointers inside them cannot be fattened.
- We identify such pointers using a whole-program points-to analysis with an *allocation-site abstraction*.

Limitations

We cannot enable a proof of memory safety in the presence of:

- External code that access program memory
- Non-standard pointer tricks (e.g., XOR-linked lists)
- Reuse of memory for different types (except via `unions`)
- Anything else that interacts poorly with fat pointers
- Concurrency (race conditions can cause memory corruption)

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
- 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.

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`.

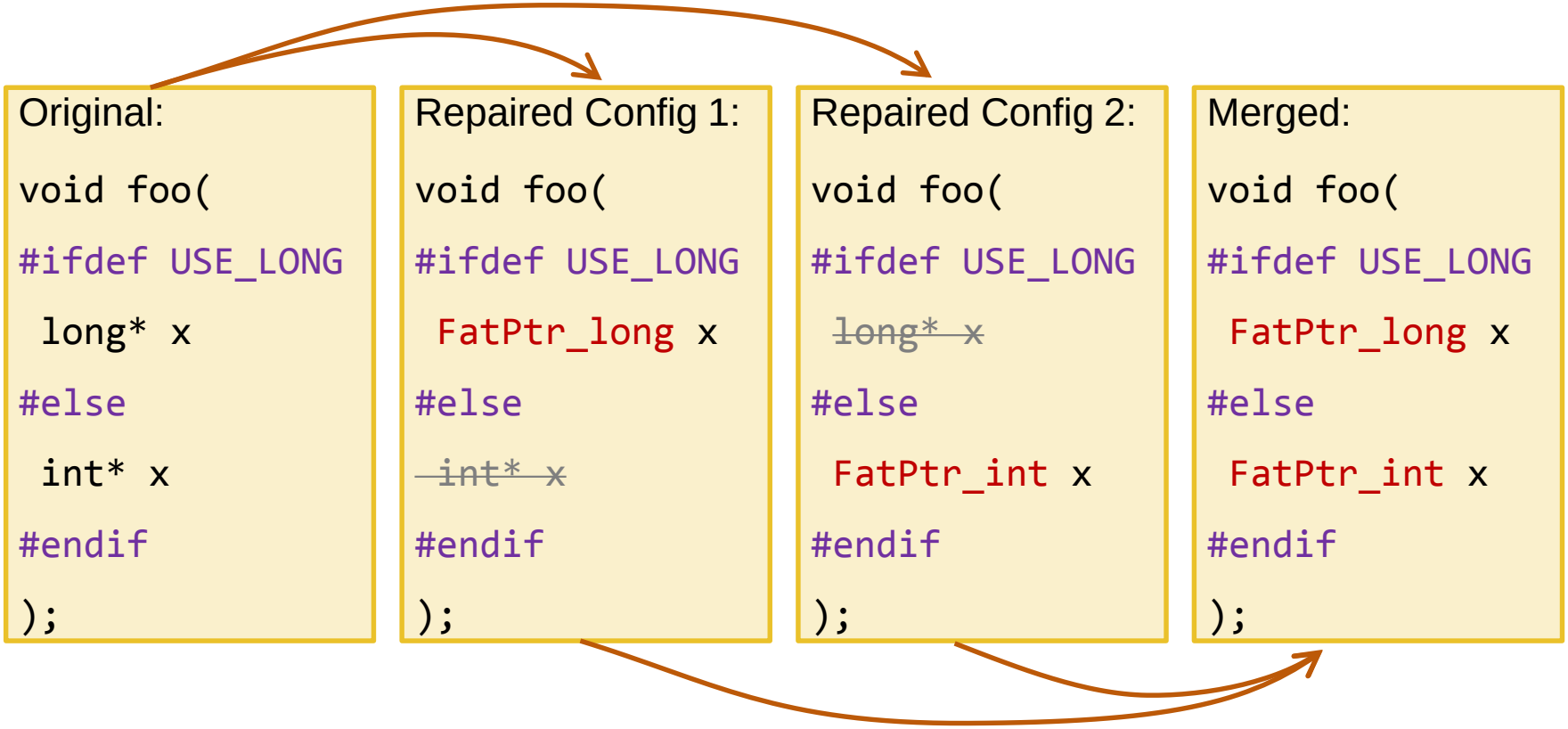
Examples of options specified in a configuration:

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

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

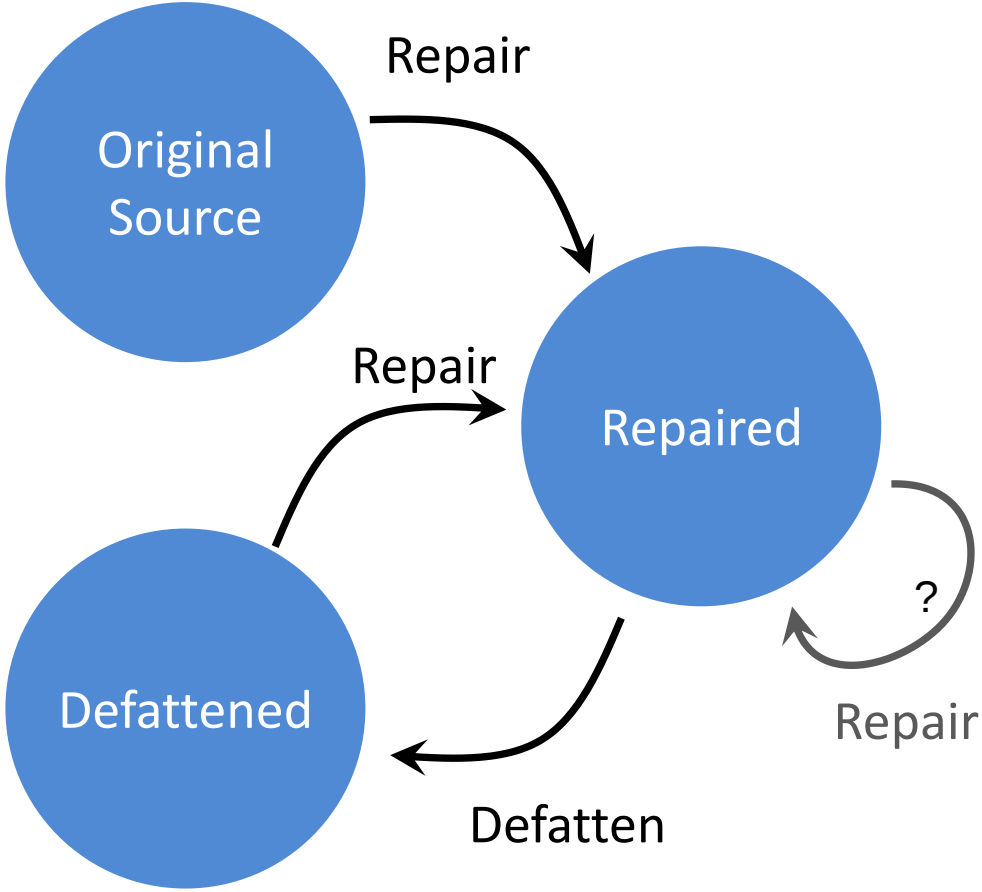


Idempotence and Defattening

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

Not yet.

But: (repair ∘ defatten) is idempotent.



Reducing runtime overhead

- Our overhead time has been around 50%. (Specifically, on bzip2.)
  - This is too high for many practical uses.
  - Can we reduce it significantly while still enabling a proof of memory safety?
    - Probably not, but automated repair is valuable even if it fixes only the most likely 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.
  - One popular commercial static analyzer gives only 1 warning for bzip2. This reduces the overhead to **6%**. (This remaining overhead is mainly due to the pointers still getting fattened even though they are never bounds-checked.)
- Next steps:
  - Refrain from fattening variables and fields if the bounds metadata isn't needed for bounds checks.
  - Performance measurements on old versions of codebases with known vulnerabilities that static analyzers warn about.