Automated Code Repair to Ensure Memory Safety

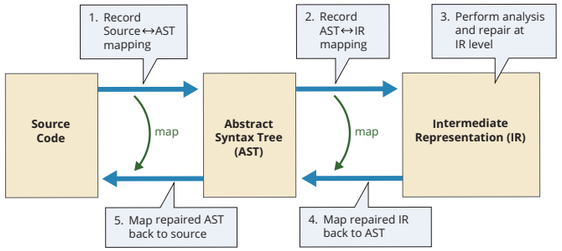
Problem

Software vulnerabilities constitute a major threat to DoD. Memory violations are among the most common and most severe types of vulnerabilities. Spatial memory vulnerabilities constitute 15% of CVEs in the NIST National Vulnerability Database and 24% of critical-severity CVEs.

Solution

We developed and implemented a technique to automatically repair source code to assure spatial memory safety. Our tool inserts code to abort the program (or call user-specified error-handling code) immediately before a memory violation would occur, preventing exploitation by attackers.

The main technique that we use (fat pointers) has been previously researched to repair code as part of the compilation process. Our work is novel in applying it as a source-code repair, which poses the difficulty of translating the repairs on the intermediate representation (IR) back to source code. The pipeline is shown below:



Intended Impact

With further development, this technology can be used by DoD to ensure memory safety as part of all software projects with code written in memory-unsafe languages (such as C and C++).

We developed an **automated technique** to repair C source code to **eliminate memory-safety vulnerabilities**.

Figure 1(a): 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;
}
```

Figure 1(b): 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;
}
```

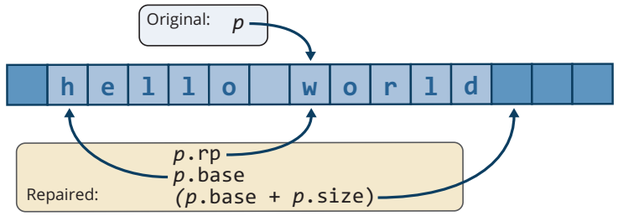
Results

The runtime overhead of our repair is around 50% on bzip2. Our DoD partners said this is too high for many of their use cases. Can we significantly reduce the overhead while still guaranteeing 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.

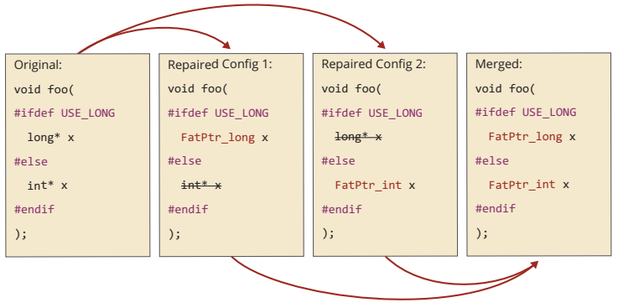
Ensuring spatial memory safety with fat pointers

Our tool replaces raw pointers with **fat pointers**, which are structs that include bounds information in addition to the pointer itself. Before dereferencing a fat pointer, a bounds check is performed. For each pointer type *T**, we define a new struct:

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



To preserve compatibility with third-party binary libraries, we identify and refrain from fattening any pointers stored in heap memory that is reachable by external binary code. The C preprocessor can include or exclude pieces of C code depending on the configuration chosen at compile time. We repair configurations separately and merge the results:



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.

DM20-0911