Automated Code Repair to Ensure Memory Safety

Memory-related bugs in C/C++ code are notorious for leading to vulnerabilities. We’re developing techniques for automated repair of source code to eliminate such vulnerabilities and enable a proof of memory safety.

What about distinguishing false alarms from true vulnerabilities?

We repair all potential memory-safety vulnerabilities, at a cost of an often small runtime overhead. (Manual tuning might be needed for performance-critical parts.)

Intermediate Representation (IR)

Problem: Static analysis generally works best on a suitable IR, but the repair must be done on the original source code.

Solution: We augment the IR with tags that record how to transform back to source.

• Each abstract syntax tree (AST) node is tagged with a reference to the corresponding original (unpreprocessed) source code text.
• We have developed a set of reversible transformations that start with the original AST and transform it to the IR.
• The IR is repaired and then transformed back to source using the tags. If a repair invalidates a tag, then the tag is ignored.

Heuristic

If a program performs arithmetic on a memory address \( p_1 \) to obtain a new memory address \( p_2 \), and \( p_2 \) is later dereferenced, then \( p_2 \) should be in the same allocated memory region as \( p_1 \).

The ISO C standard actually requires compliance with this heuristic (on pain of undefined behavior) for arithmetic on values of pointer type.

Static Analysis and Repair

For each memory access \( *p \), we generate a precondition ensuring it is within bounds:

\[
\text{MemLo}(p) \leq p < \text{MemHi}(p)
\]

where \( \text{MemLo} \) and \( \text{MemHi} \) are functions of the provenance (not value) of \( p \).

If the value of \( p \) at a particular timepoint in an execution trace was computed by pointer arithmetic on the result of a memory allocation (e.g., malloc), then \( \text{MemLo}(p) \) and \( \text{MemHi}(p) \) denote the lower bound (incl.) and upper bound (excl.) of this memory region.

For each precond, do one of the following:

• Prove that it is satisfied.
• Add bounds check with existing variables.
• Modify function signatures and/or structs to include bounds info (“fat pointers”).
• Modify program to record info about bounds in global lookup table (as in SoftBound).

In the past, fat pointers were disfavored due to inability to analyze or repair libraries available only in binary form. However, new developments in SEI’s Pharos platform will allow us to overcome this limitation by tackling binaries.

Leaks of Sensitive Data via Stale Reads

Consider a web server that stores a received request in a reusable buffer. Once the server is done with the request, the buffer holds stale data, which can later be (partially) overwritten by a new request.

Example: Reused buffer with stale data

Buffer contents after first HTTP request:

Buffer contents after second HTTP request:

We developed a heuristic for identifying such a buffer and what part of it is valid.

Definition: A buffer \( B \) is qualifying if and only if every write is to either index 0 or the successor of the last written position (LWP).

Our sequential write heuristic posits that a qualifying array contains valid (non-stale) data up to and including the LWP.

We implemented a dynamic analysis based on this heuristic, targeting C and Java. Our analysis detects JetLeak (CVE-2015-2080) in Jetty and Heartbleed in OpenSSL.

In analyzing GNU coreutils (80k LOC, plus 486k LOC of library), there were 17 alarms (all suspected/confirmed false positives).

Developing a static analysis is future work.