

Automated Code Repair

Will Klieber

December 2018



Copyright 2018 Carnegie Mellon University. All Rights Reserved.

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.

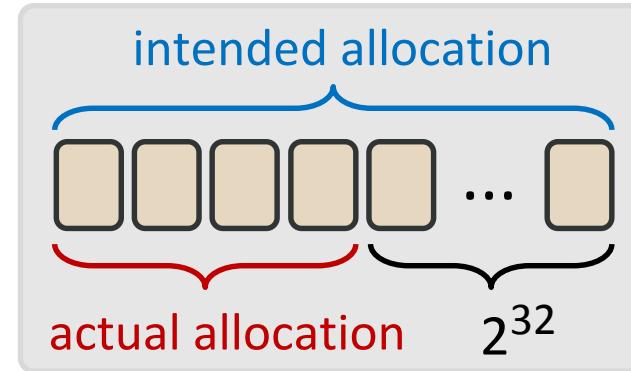
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

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.

DM18-1391

Premises for Automated Code Repair (ACR)

1. Many security bugs follow common patterns.
 - E.g., “`p = malloc(n * sizeof(T))`” where n is attacker-controlled.
 - Integer overflow \Rightarrow too little memory gets allocated.
2. By recognizing such a pattern, it is possible to confidently guess the developer's intention (the *inferred specification*).
 - E.g., “Try to allocate enough memory to hold n objects of type T ”.
3. It is possible to repair the code to satisfy this inferred specification.
 - Check if overflow occurs; if so, simulate `malloc` failing (by returning NULL).



What about false positives? Okay to ‘repair’ them, if we don’t break code. Performance-critical parts might require manual review.

Why not repair as a compiler pass? Requires permanent change to build process, and it is very hard to audit the results.

Repair of integer overflow in array bounds check

Repair: **UADD(start, n)** /* UADD is defined below */

```
if (start + n <= dest_size) {  
    memcpy(&dest[start], src, n);  
} else {  
    return -EINVAL;  
}
```

Example: copy n bytes from src to $dest$, starting at index $start$ of $dest$.

```
1. inline static size_t UADD(size_t lop, size_t rop) {  
2.     size_t result; bool overflow;  
3.     overflow = __builtin_add_overflow(lop, rop, &result);  
4.     if (overflow) {result = SIZE_MAX;}  
5.     return result;  
6. }
```

Inferred specification:
inequality comparisons
involving array indices or bounds should behave as in normal arithmetic (not modular arithmetic).
Normal arithmetic can be emulated using **saturation arithmetic**.

Experimental results for repair of integer overflows

Xi Wang et al., OSDI 2012 -- uses SMT solver Boolector

	Overflows reported by Kint	Overflows that are sensitive	Overflows fully repaired
OpenSSL (1.0.2g)	969	233	180 (77%)
Jasper (1.900.1)	481	101	53 (52%)

An overflow is **sensitive** if it involves variables associated with array indices or bounds.

Some of the 'repairs' are actually false positives (i.e., operation never overflows). Then our 'repair' just adds a little overhead. **It never breaks working code.**

Other repairs fix known vulnerabilities.

Inserting missing bounds checks

Ultimate goal: Repair program to enable proof of memory safety.

When the result of pointer arithmetic is used to access memory, ensure that it is still within the bounds of the original allocated region.

```
v = calloc(n, sizeof(int));  
...  
scanf("%i", &k);  
for (i=0; i < k; i++) {  
    if (v[i] < 0) break;  
    v[i] += 1;  
}
```

Repair: Insert bounds check

```
if (i >= n) abort();
```