UNDER-CONSTRAINED SYMBOLIC EXECUTION: CORRECTNESS CHECKING FOR REAL CODE

DAVID A. RAMOS AND DAWSON ENGLER
STANFORD UNIVERSITY

24TH USENIX SECURITY SYMPOSIUM
AUGUST 12, 2015
CONTRIBUTIONS

- Technique + tool for finding deep bugs in real, open source C/C++ code
  - No manual testcases
  - No functional specification
- Bugs reported **may** have security implications; **exploitability** must be determined manually
  - Memory access, heap management, assertion failures, division-by-zero
- Found 77 new bugs in BIND, OpenSSL, Linux kernel
  - 14 Linux kernel vulnerabilities (mostly minor DoS issues)
MOTIVATION: CURRENT PRACTICE

- Code reviews
- “Safer” languages
- Manual (regression) testing
- Static analysis (Coverity, clang static analyzer, etc.)

Bugs are everywhere!
SYMBOLIC EXECUTION

• Provide *symbolic* rather than *concrete* inputs

• Conceptually: explore **all** paths through a program

• Accurately track all memory values (bit precision)

• Report paths/inputs that crash
  ▸ Generate concrete testcase

• KLEE tool (prior work: OSDI 2008)
EXAMPLE

int foo(int x) {
    if (x)
        return x/10;
    else
        return 10/x;
}
int foo(int x) {
  if (x) return x/10;
  else return 10/x;
}

symbolic branch
```c
if (x)  
    return x/10;
else  
    return 10/x;
```

**State 1**

**State 2**

**Division: OK**
Division: ERROR
PROBLEM: SCALABILITY

• Path explosion
  ‣ $|\text{paths}| \sim 2 |\text{if-statements}|$

• Path length and complexity
  ‣ Undecidable: infinite-length paths (halting problem)

• SMT query complexity (NP-complete)
SOLUTION: UNDER-CONSTRAINED

• Directly execute individual functions within a program
  ‣ Less code = Fewer paths
  ‣ Function calls executed (inter-procedural)
  ‣ Able to test previously-unreachable code

• Challenges
  ‣ Complex inputs (e.g., pointer-rich data structures)
  ‣ Under-constrained: inputs have unknown preconditions
    - False positives
UC-KLEE TOOL

- Extends KLEE tool (OSDI 2008)
- Runs LLVM bitcode compiled from C/C++ source
- Automatically synthesizes complex inputs
  - Based on lazy initialization (Java PathFinder)
  - Supports pointer manipulation and casting in C/C++ (no type safety)
  - User-specified input depth (k-bound) [Deng 2006]
LAZY INITIALIZATION

• Symbolic (input) pointers initially **unbound**

• On first dereference:
  ‣ New object allocated
  ‣ Symbolic pointer **bound** to new object’s address
  ‣ Assume no aliasing (i.e., no cyclical data structures)

• On subsequent dereferences:
  ‣ Pointer resolves to object allocated above
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

**EXAMPLE**

- `n == 0`: Node is null.
- `n != 0`: Node is non-null.
- `n == &uc_node1`: Node is equal to `uc_node1`.

```
struct uc_node {
    int val;
    struct uc_node* next;
};
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
EXAMPLE

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
USE CASES

- Equivalence checking: patches
  - Yesterday’s code vs. today’s code (i.e., fewer bugs today)
  - Goal: detect (and prevent!) new crashes introduced by patches
  - Other uses discussed in CAV 2011 paper
PATCHES

99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.
127 little bugs in the code...

Source: https://twitter.com/phabricator
USE CASES

• Equivalence checking: patches
  ‣ Yesterday’s code vs. today’s code (i.e., fewer bugs today)
  ‣ Goal: detect (and prevent!) new crashes introduced by patches
  ‣ Other uses discussed in CAV 2011 paper

• General bug-finding: rule-based checkers
  ‣ Single version of a function; under-constrained + additional checker rules
  ‣ Memory leaks, uninitialized data, unsafe user input
  ‣ Simple interface for adding new checkers
EQUIVALENCE CHECKING

retA = fooA(x);
retB = fooB(x);
assert(retA == retB);

identical input (symbolic)

assert equivalence
EQUIVALENCE CHECKING

• Value equivalence
  ‣ Return value
  ‣ Arguments passed by reference
  ‣ Global/static variables
  ‣ System call effects (modeled)

• Error (crash) equivalence
  ‣ Both versions typically have the same same (unknown) preconditions!
  ‣ Neither version crashes on an input
  ‣ Both versions crash on an input

USE CASE: whether patches introduce **crashes**
EQUIVALENCE CHECKING

• Check per path equivalence of two functions

• If all paths exhausted, equivalence verified (up to input bound)
EVALUATION

- BIND, OpenSSL
  - Mature, security-critical codebases (~400 KLOC each)
- Patches
  - BIND: 487 patches to 9.9 stable (14 months)
  - OpenSSL: 324 patches to 1.0.1 stable (27 months)
- Ran UC-KLEE for 1 hour on each patched function
EVALUATION: PATCHES

• Discovered **10 new bugs** (4 in BIND, 6 in OpenSSL)
  ‣ 2 OpenSSL DoS vulnerabilities:
    - CVE-2014-0198: NULL pointer dereference
    - CVE-2015-0292: Out-of-bounds `memcpy` read

• Verified (w/ caveats) that patches do not introduce crashes
  ‣ 67 (13.8%) for BIND, 48 (14.8%) for OpenSSL
  ‣ Caveat: max. input size (25KB), tool limitations/bugs
OPENSSL CVE-2014-0198

```c
void do_ssl3_write()
{
    if (wb->buf == NULL)
        if (!ssl3_setup_write_buffer(s))
            return -1;

    /* If we have an alert to send, lets send it */
    if (s->s3->alert_dispatch) {
        i=s->method->ssl_dispatch_alert(s);
        if (i <= 0)
            return(i);
    /* if it went, fall through and send more stuff */
    }

    /* If we have an alert to send, lets send it */
    if (s->s3->alert_dispatch) {
        i=s->method->ssl_dispatch_alert(s);
        if (i <= 0)
            return(i);
    /* if it went, fall through and send more stuff */
    }

    unsigned char *p = wb->buf;
    *(p++)=type&0xff;
    *(p++)=type&0xff;
}
```
OPENSSL CVE-2014-0198

• Uncommon code path
  ‣ SSL_MODE_RELEASE_BUFFERS runtime option (used by Apache mod_ssl)
  ‣ SSL alert pending (could be triggered by attacker)
  ‣ Difficult to consider this case with traditional testing
FALSE POSITIVES

• Function’s inputs have unknown **preconditions**

• Partial solutions
  ▸ Automated heuristics
  ▸ Manual annotations (lazily, as needed)
    - Written in C/C++, separate from codebase
    - Simple annotation can silence **many** errors
FALSE POSITIVES: EXAMPLE (BIND)

```c
int isc_region_compare(isc_region_t *r1, isc_region_t *r2) {
    unsigned int l;
    int result;

    REQUIRE(r1 != NULL);
    REQUIRE(r2 != NULL);

    l = (r1->length < r2->length) ? r1->length : r2->length;

    if ((result = memcmp(r1->base, r2->base, l)) != 0)
        return ((result < 0) ? -1 : 1);
    else
        return ((r1->length == r2->length) ? 0 :
                (r1->length < r2->length) ? -1 : 1);
}

INVARIANT(r->length <= OBJECT_SIZE(r->base));

623 errors silenced (7.5% of all errors reported for BIND)
```
MANUAL ANNOTATIONS

• BIND: 400 lines of annotation code (~0.1%)
• OpenSSL: 60 lines of annotation code (~0.02%)
• Reasonable effort relative to code size (~400 KLOC) and importance
GENERAL BUG-FINDING

• Run **single version** of a function (w/ lazy initialization)

• Individual checkers look for specific types of bugs:
  - Leak checker
  - Uninitialized data checker
  - User input checker

• Like Valgrind but applied to **all execution paths**
EVALUATION

• 20,000+ functions: BIND, OpenSSL, Linux kernel (~12 MLOC)
• Found 67 new bugs
  ▸ 37 memory leaks
    - Linux kernel: exploitable AUTH_GSS leak in NFS SunRPC layer
  ▸ 19 uses of uninitialized data
    - BIND: DNS UDP port PRNG selected by uninitialized value
    - Linux kernel: leak of private kernel stack data via firewire ioctl
  ▸ 11 unsafe user input (Linux kernel only)
    - VMware VMCI driver: unchecked memcpy length (~Heartbleed)
    - CEPH distributed file system: division-by-zero (kernel FPE)
USER INPUT CHECKER

- User input is fully-constrained (an attacker may supply any value); no unknown input preconditions
- Checker tracks whether each symbolic byte is UC/FC
- Checker emits UNSAFE_INPUT flag if error is caused by FC input
- Suppresses flag for inputs possibly sanitized (false pos. trade-off)
- C annotations: specify functions returning user input
  - Linux: `get_user`, `copy_from_user`, syscall args
  - BIND: `isc_buffer_getuint8`
  - OpenSSL: byte-swaps (n2s, n2l, etc.) [Chou]
KERNEL VMCI VULNERABILITY

Fully constrained

```c
static int dg_dispatch_as_host(..., struct vmci_datagram *dg) {
    dg_size = VMCI_DG_SIZE(dg);
    ...
    dg_info = kmalloc(sizeof(*dg_info) + (size_t) dg->payload_size, GFP_ATOMIC);
    ...
    memcpy(&dg_info->msg, dg, dg_size);
    ...
}
```

Send up to 69,632 bytes from host private kernel memory to guest OS

Similar to Heartbleed! (much lower impact)

Unchecked `memcpy` length

`copy_from_user()`
CONCLUSION

• Under-constrained symbolic execution
• Equivalence checking: patches
• General bug-finding: rule-based checkers
• Experimental results: BIND, OpenSSL, Linux kernel
QUESTIONS?

@ramosbugs