DEBUGGING: DYNAMIC PROGRAM ANALYSIS

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification

System Invariants

properties of a program must hold over the entire run:

- integrity of data
- no access of data of other processes
- handling of mathematical exceptions
- adhere to privileges

$\Rightarrow$ ensured by operating system and hardware

- security for user
- valuable information for debugging

Example: Heap Memory Problems

common source of errors in C and C++ programs: misuse of heap memory

example:

```
   ...
1    char *s = malloc(24);
2    char *t = malloc(20);

3    strcpy(t, "hello");
4    s[33] = 'b';

5    printf ("%s\n", t);
6    free (t);
7    free (t);
   }
```

problems:

- no `free (s)`
- write in line 4
- `t` is released twice

Software Memory Problems

typical problems related with memory allocation / deallocation:

- use of uninitialized memory ($\rightarrow$ undefined values)
- accessing memory that has been released
- accessing memory beyond allocated block
- accessing inappropriate stack areas
- memory leaks
- passing uninitialized memory to system calls

symptoms:

- no observable problem
- increasing memory consumption
- segmentation fault

$\Rightarrow$ memory debugging

Dynamic Binary Analysis

- analysis of program's code at runtime
- augmentation of original code with analysis code
  = **dynamic binary instrumentation** (DBI)

$\Rightarrow$ analysis code runs in parallel with original code

$\Rightarrow$ useful information about the program

applications:

- error detection
- profiling

DBI: Pros and Cons

advantages:

- user-friendliness (no modification of code)
- no recompilation, no relinking
- 100 % coverage
- easy to extend (add plugins to core analysis framework)

problems:

- loss in performance
- data and operations in analysis tool are as complex as in the original program
- false positives

Valgrind

- instrumentation framework for building dynamic analysis tools
- program under observation is executed on synthetic CPU together with instrumentation code
- some Valrind-based tools:
  - memory error detector (**memcheck**)
    - normal call of program: `myprog arg1 arg2`
    - with Valgrind:
      `valgrind -leak-check=yes myprog arg1 arg2`
  - thread error detectors
  - cache and branch-prediction profiler
  - call-graph generating cache and branch-prediction profiler
  - heap profiler

Memcheck

- detection of memory-realted errors in C and C++ programs
- default tool of Valgrind

example output:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

categories of leaks:

- definitivly lost
- probably lost

Example: Buggy Shell-Sort (1/3)

```
...
 3  static void shell_sort (int a[], int size) {
      ...

16      for (i = h; i < size; i++) {
17          int v = a[i];

18        for (j = i; j >= h && a[j - h] > v; j -= h)
19            a[j] = a[j - h];
      ...

26  int main (int argc, char *argv[]) {
      ...
35    shell_sort(a, argc);
```

JKU

Example: Buggy Shell-Sort (2/3)

Valgrind on shell-sort with defect:

```
$ valgrind  ./shell 11 14
==16992== Memcheck, a memory error detector
==16992== ...
==16992== Invalid read of size 4
==16992==    at 0x4006A6: shell_sort (shell.c:17)
==16992==    by 0x4007D6: main (shell.c:35)
==16992==  Address 0x40EE902C is 0 bytes after a block of size 8 alloc'd
==16992==    at 0x4C2DC10: malloc (vg_replace_malloc.c:299)
==16992==    by 0x40076F: main (shell.c:31)
==16992==
==16992== Invalid write of size 4
==16992==    at 0x4006E0: shell_sort (shell.c:19)
==16992==    by 0x4007D6: main (shell.c:35)
==16992==  Address 0x40EE902C is 0 bytes after a block of size 8 alloc'd
==16992==    at 0x4C2DC10: malloc (vg_replace_malloc.c:299)
==16992==    by 0x40076F: main (shell.c:31)
Output: 0 11
```

JKU

Example: Buggy Shell-Sort (3/3)

Valgrind on shell-sort with fix (`shell_sort(a, argc-1);`) :

```
valgrind  ./shell 11 14
==17395== Memcheck, a memory error detector
==17395== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==17395== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==17395== Command: ./shell 11 14
==17395==
Output: 11 14
==17395==
==17395== HEAP SUMMARY:
==17395==     in use at exit: 0 bytes in 0 blocks
==17395==   total heap usage: 2 allocs, 2 frees, 1,032 bytes allocated
==17395==
==17395== All heap blocks were freed - no leaks are possible
==17395==
==17395== For counts of detected and suppressed errors, rerun with: -v
==17395== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

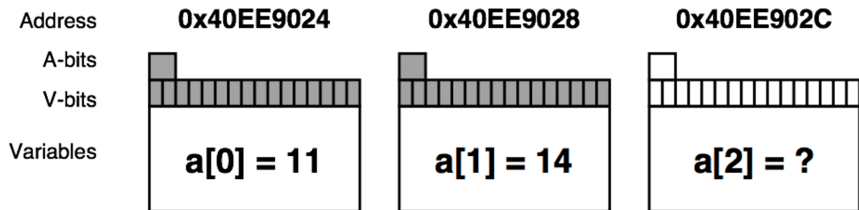
JKU

Shadow Values

- shadow every value with another value that describes it
- writing and propagation together with values of original program
- no change of original program behavior

original operations	shadow operations
int p = malloc (...)	shadow (p) = undef
R1 = 1	shadow (R1) = def
R1 = R2	shadow (R1) = shadow (R2)
R1 = R2 + R3	R1 = add$_{sh}$ (R2, R3)
if (R1 == 0) ...	error if R1 is undef

Shadow Memory



- V-bit set: corresponding bit is initialized
- A-bit set: corresponding byte is accessible

JKU

Valid-Value Bits (V-Bits)

- every processed bit has an associated valid-value bit (V-bit)
  $\Rightarrow$ each byte has 8 V-bits
- V-bit indicates if value in original bit is valid
- V-bit is checked immediately on operations on a value that affect program's externally-visible behavior
  $\Rightarrow$ error if value is undefined and used for generating an address, making a control flow decision, or as argument for a system call
- simple read accesses to uninitialized memory do not result in warnings:

```
struct S { int x; char c; };
  struct S s1, s2;
  s1.x = 42; // 5 bytes init
  s1.c = 'z';
  s2 = s1; // 8 bytes copied, no warning
```

Access-Bits (A-Bits)

- all global data is marked “accessible” on program start (= A-bits are set)
- `malloc()` sets A-bits for the area returned
- `free()` clears them
- local variables are “accessible” on function entry and “non-accessible” on exit

$\Rightarrow$ error if “non-accessible” data is used

Address Sanitizer: Overview

- compile-time instrumentation module
- run-time library: Linux, OS X, Android, Windows
- supported by LLVM and GCC

example:

```
int global_array[100] = {1};
int main(int argc, char **argv) {
  return  global_array[argc + 100];
}
```

- `clang -fsanitize=address test.c -o test`
- output when running `./test`

```
==28580==ERROR: AddressSanitizer: global-buffer-overflow on address 0x0000
READ of size 4 at 0x000000719cd4 thread T0
...
0x000000719cd4 is located 4 bytes to the right of global variable
'global_array' defined in 'test4.c:1:5' (0x719b40) of size 400
```

Coverage for Debugging

detecting anomalies by using coverage information:

1. Every failure is caused by an infection, which again is caused by a defect.
2. Teh defect must be executed in order to trigger the infection.
3. Thus, code that is executed only in failing runs is more likely to contain the defect than the code that is always executed.

⇒ use coverage tool that keeps track of executed code lines

⇒ compare coverage of passing and failing run

Example: Coverage

```
     int middle(int x, int y, int z) {
 1   int m = z;
 2   if (y < z) {
 3      if (x < y)
 4          m = y;
 5      else if (x < z)
 6        m = y;
     } else {
 7   if (x > y)
 8        m = y;
 9   else if (x > z)
10        m = x;
     }
11   return m;
   }
```

x	3	1	3	5	5	2
y	3	2	2	5	3	1
z	5	3	1	5	4	3
1	●	●	●	●	●	●
2	●	●	●	●	●	●
3	●	●			●	●
4		●				
5	●				●	●
6	●					●
7			●	●		
8			●			
9				●		
10						
11	●	●	●	●	●	●
	ok	ok	ok	ok	ok	fail

⇒ consider line 6, because it has been exectued by only one passing test

JKU

Challenges in Parallel Programming

... so far we considered only sequential programs

with parallel programs we have the same challenges as with sequential programs plus

- ensure correctness of overall program
- ensure correctness of n parallel processes
- new problems
  - deadlock
  - race condition
  - irreproducibility

Deadlocks

typical deadlock situation:

- two threads T_1 and T_2
- shared resources: m_1 and m_2
- access of resources:
  - T_1 waits to synchronize with T_2 on m_1, but ...
    m_1 can only be established by T_2 after m_2
  - T_2 waits to synchronize with T_1 on m_2, but ...
    m_2 can only be established by T_1 after m_1

$\Rightarrow$ **deadlock** (usually system freezes)

Finding Deadlocks

- models
  - either build or extract abstract model
  - model checking or unit testing
  - goal is exhaustive simulation of all schedules
- search for cyclic dependencies
  - priority inversion (static lock/mutex order)
  - cycles in lock dependency graph
- generate massif load, insert jitter
  - wait for random time between locks/unlocks
  - add artificial work

Debugging Deadlocks

- access to program state of all threads
  - either through debugging/logging thread
  - with symbolic debuggers
- attaching symbolic debuggers
  - after program seemed to be frozen
  - `gdb program pid`
  - threads, thread 2, bt
- external tools that monitor locking order, e.g., helgrind (which uses sand boxing)
- programming discipline: proper lock protection

Happens-Before Relation

- dependency between events
- events in the same thread/process ordered by execution order
- synchronization among threads/processes
  - sending/receiving message
  - locking/unlocking (of one particular lock)
  - waiting for a condition/enabling a condition

shared access events should be ordered by happens before relation, otherwise

- data races
- non-deterministic behavior

Example: Lock Protection

proper lock protection

```
Thread 1                Thread 2

lock (mu);
v = v + 1;
unlock (mu);
                        lock (mu);
                        v = v + 1;
                        unlock (mu);
```

improper lock protection

```
Thread 1                Thread 2

y = y + 1;
lock (mu);
v = v + 1;
unlock (mu);
                        lock (mu);
                        v = v + 1;
                        unlock (mu);
                        y = y + 1;
```

Example: Data Race

```
nclude <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
   var++; /* this is line 6 */
   return NULL;
}

int main () {
   pthread_t child;
   pthread_create(&child, NULL, child_fn, NULL);
   var++;  /* this is line 13 */
   pthread_join(child, NULL);
   return 0;
}
```

Example: Data Race

```
Thread #1 is the program's root thread

Thread #2 was created
  ...
  by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
   at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x4005DC: child_fn (simple_race.c:6)
   ...
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)

Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```