Dynamic Analysis

Lecture 8: CPEN 400P

Karthik Pattabiraman, UBC

(Some slides are based on Wes Weimer's EECE481 at Univ. of Michigan, and some slides on "Software Analysis and Testing" book by Mauro Pezze and Michal Young, Chapter 19)

Outline

What’s dynamic analysis ?

Pros and Cons of dynamic analysis

Steps in dynamic analysis

Example 1: Memory Tracking

Example 2: Race Detection

Dynamic Analysis

Executing a program with one or more (chosen) inputs and observing its execution

- Testing: Check if program produces correct output or not (Not covered)

- Fuzzing: Give random/arbitrary inputs and test if program crashes (next class)

- Check if specific properties or conditions are violated at runtime (this class)

Testing

Most commonly used dynamic analysis technique in industry

Different forms of testing

- Unit testing
- Integration testing
- Black box testing
- White box testing
- Performance testing
- Interaction testing

“Testing can only reveal the presence of bugs, not their absence” - E. W. Djkstra

Fuzzing

Feed random inputs to the program to find crash-causing inputs

Programs are surprisingly fragile to unexpected inputs

Different forms of fuzzing

- Mutation
- Evolution
- Generation
- Model-based

Outline

What’s dynamic analysis ?

Pros and Cons of dynamic analysis

Steps in dynamic analysis

Example 1: Memory Tracking

Example 2: Race Detection

Why Dynamic Analysis ?

Can analyze the actual execution of the program and obtain runtime values

Much more **precise** compared to static analysis

- No need to over-approximate across all paths
- No need to worry about scalability of the analysis
- No need to be conservative when the analysis cannot determine a value

Many common code constructs are difficult to statically analyze (e.g., threads)

Can take external environment into account (e.g., interrupts, I/O operations etc.)

Does not need the source code of libraries or 3rd party component

Why NOT Dynamic analysis ?

Code instrumentation is often needed

- Incurs runtime overhead (can be substantial for certain kinds of analysis)
- Can result in bugs being masked (Heisenbugs), e.g., race conditions
- Can be challenging to instrument external libraries or 3rd party code

Dynamic analysis is NOT **complete**

- Cannot analyze all paths in the program, only those that are executed
- Limited by the representativeness of test inputs (w.r.p. To real-world inputs)
- Difficult to get the program to exercise corner cases, esp. Failure handling
- Cannot typically obtain global program information across large scopes

Example of Static Vs. Dynamic Analysis

```
int fd;

int *getval(void)

{

  int *tmp;

  read(fd, &tmp, sizeof(tmp));

  return tmp;

}

void foo(void)

{

  int *b = getval();

  *b = 0;

}
```

Source:*https://www.embedded.com/static-vs-dynamic-analysis-for-secure-code-development-part-2/*

Code on the right will throw a NP Exception (why?)

Static analysis cannot easily find the error

- Overly conservative (mark all values of tmp as potentially Null)
- Overly optimistic (ignore potential Null values of *tmp)*
- Need for *inter-procedural* analysis

Dynamic analysis can find the error easily

- Instrument the file read function
- Check value of tmp after read
- Raise alert if null (as it’s dereferenced)

Outline

What’s dynamic analysis ?

Pros and Cons of dynamic analysis

Steps in dynamic analysis

Example 1: Memory Tracking

Example 2: Race Detection

Dynamic Invariant Deduction

Steps in Dynamic Analysis

1. Instrument the program to capture the “property of interest”

2. Execute the program under one or more inputs in a controlled environment

3. Monitor program for the property violation at the instrumented sites

4. (Optional) Use static analysis results to optimize the dynamic execution monitoring

Step 1: Instrument the program

Instrumenting a program involves modifying or rewriting its source code or binary executable to change its behavior, typically to record additional information.

e.g., add print(“reached line $X”) to each line X

This can be done at compile time

e.g., gcov, cobertura, LLVM, etc.

It can also be done via a specialized VM

e.g., Valgrind, specialized JVMs, PIN etc.

Step 2: Execute the program

Need to choose inputs that are representative of its real execution, as well as corner-case inputs

Need to ensure that as many paths are covered as possible at runtime

Can be done in a VM or other controlled environment in case of vulnerabilities

- Especially when bugs or vulnerabilities can cause harm to the system
- Overhead of execution can be very high

Step 3: Monitor the program for property violations

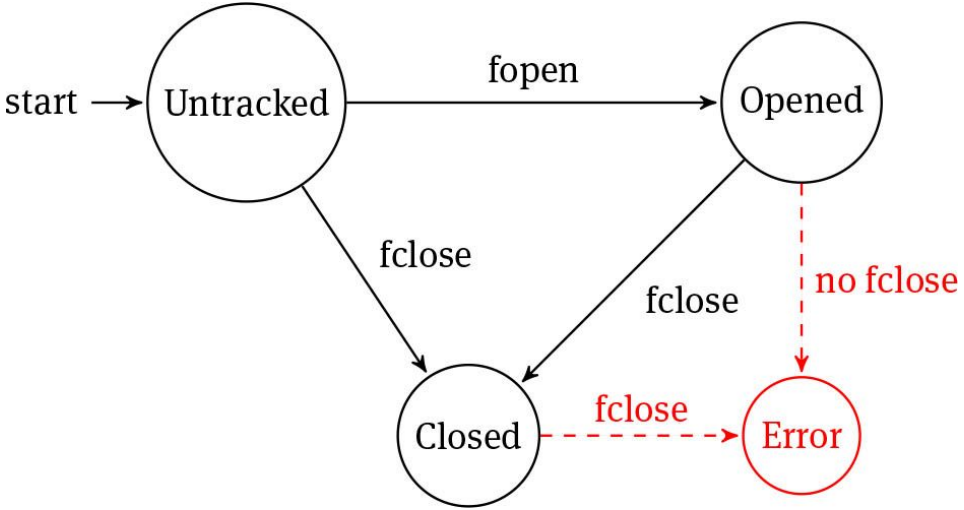
Check for property violations at runtime

Prefer efficient structures like state machines

- Most properties can be expressed via such abstractions
- Use static analysis to optimize for monitoring only what's essential
- Instrumentation should correspond to the property being tracked

Upon violation of the invariant

- Raise an alert and stop the program



File open/close state machine at runtime - tracks double closing and not closing files

Step 4: Optimize monitoring using static analysis

Goal: To reduce the amount of instrumentation and monitoring by ruling out obvious false positives. Examples are:

- When looking for heap buffer overflows, no need to monitor pointer writes to non-heap objects
- When monitoring for deadlocks and race conditions, no need to monitor sections of code that only single threads will ever execute
- When monitoring for null pointer exceptions, no need to monitor pointers that are allocated and never reassigned (unless the allocation fails)

Some optimizations may require both static and dynamic analysis

Outline

What’s dynamic analysis ?

Pros and Cons of dynamic analysis

Steps in dynamic analysis

Example 1: Memory Tracking

Example 2: Race Detection

Memory Tracking: Problem

Buffer overflows are a big problem in C/C++

- Writing past the end of a buffer can lead to undefined values
- Lead to overwriting of security critical data (e.g., return addresses, variables)
- One of the biggest causes of security vulnerabilities in the field

Focus on heap buffer overflows caused by copying - buffers allocated with malloc

Root cause: Buffer size is lesser than the length of the data copied to it

Memory Tracking: Example

```
…
int main (int argc, char *argv[]) {
        char sentinel_pre[] = "2B2B2B2B2B";
        char subject[]  = "AndPlus+%26%2B+%0D%";
   char sentinel_post[] = "26262626";
   char *outbuf = (char *) malloc(10);
   int return_code;

        printf("First test, subject into outbuf\n");
   return_code = cgi_decode(subject, outbuf);
   printf("Original: %s\n", subject);
   printf("Decoded:  %s\n", outbuf);
   printf("Return code: %d\n", return_code);

        printf("Second test, argv[1] into outbuf\n");
        printf("Argc is %d\n", argc);
   assert(argc == 2);
   return_code = cgi_decode(argv[1], outbuf);
   printf("Original: %s\n", argv[1]);
   printf("Decoded:  %s\n", outbuf);
   printf("Return code: %d\n", return_code);
}…
```

Output parameter of fixed length
Can overrun the output buffer

Running Purify Tool on the Example

```
[I] Starting main
[E] ABR: Array bounds read in printf (1 occurrence)
    Reading 11 bytes from 0x00e74af8 (1 byte at 0x00e74b02 illegal)
    Address 0x00e74af8 is at the beginning of a 10 byte block
    Address 0x00e74af8 points to a malloc'd block in heap 0x00e70000
    Thread ID: 0xd64
...
[E] ABR: Array bounds read in printf (1 occurrence)
    Reading 11 bytes from 0x00e74af8 (1 byte at 0x00e74b02 illegal)
    Address 0x00e74af8 is at the beginning of a 10 byte block
    Address 0x00e74af8 points to a malloc'd block in heap 0x00e70000
    Thread ID: 0xd64
...
[E] ABWL: Late detect array bounds write (1 occurrence)
    Memory corruption detected, 14 bytes at 0x00e74b02
    Address 0x00e74b02 is 1 byte past the end of a 10 byte block at 0x00e74af8
    Address 0x00e74b02 points to a malloc'd block in heap 0x00e70000
    63 memory operations and 3 seconds since last-known good heap state
    Detection location - error occurred before the following function call
          printf          [MSVCRT.dll]
...
        Allocation location
          malloc          [MSVCRT.dll]
...
[I] Summary of all memory leaks... (482 bytes, 5 blocks)
...
[I] Exiting with code 0 (0x00000000)
    Process time: 50 milliseconds
[I] Program terminated ...
```

Instrumentation

- Instrument program to trace memory access
  - record the state of each memory location
  - detect accesses incompatible with the current state
  - Attempts to access unallocated memory
- Read from uninitialized memory locations
  - array bounds violations:
- Add memory locations with state *unallocated* before and after each array
- Attempts to access these locations are detected immediately

State Machine to Track Buffer Overflows

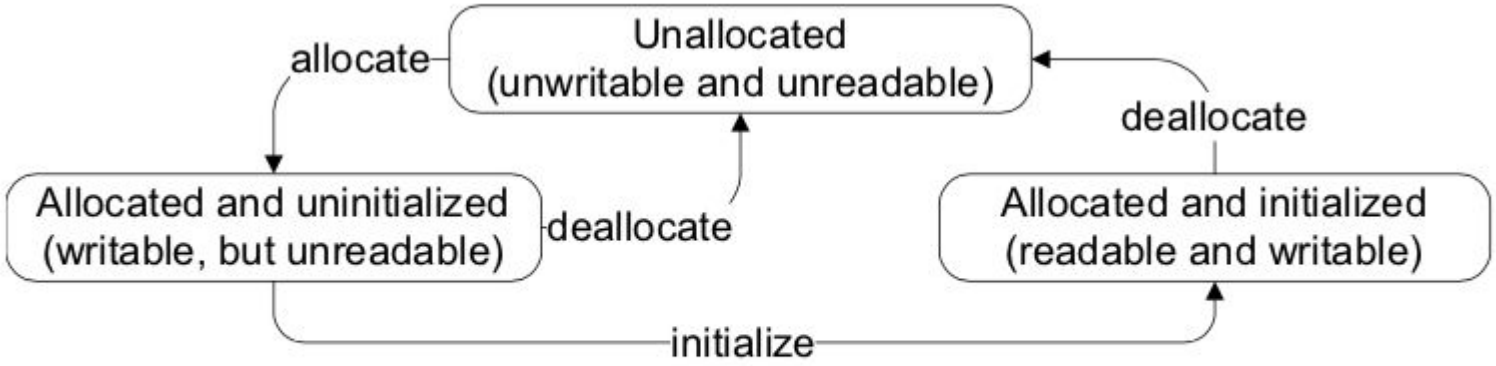


Figure 19.3: States of a memory location for dynamic memory analysis (adapted from Hastings and Joyce [HJ92]).

Outline

What’s dynamic analysis ?

Pros and Cons of dynamic analysis

Steps in dynamic analysis

Example 1: Memory Tracking

Example 2: Race Detection

Dynamic Invariant Deduction

Data Races

Testing: not effective (nondeterministic interleaving of threads)

Static analysis: computationally expensive, and approximated

Dynamic analysis: can amplify sensitivity of testing to detect potential data races

–avoid pessimistic inaccuracy of finite state verification

–Reduce optimistic inaccuracy of testing

Is there a data-race in this example ?

```
// Thread #1
while (true) {
        lock(mutex);
        v = v + 1;
        unlock(mutex);
        y = y + 1;
}
```

```
// Thread #2
while (true) {
        lock(mutex);
        v = v + 1;
        unlock(mutex);
        y = y + 1;
}
```

How about in this example ?

```
// Thread 1
while (true) {
        lock(mu1);
        v = v + 1;
        unlock(mu1);
        lock(mu2);
        v = v + 1;
        unlock(mu2);
}
```

```
// Thread 2
while (true) {
        lock(mu1);
        v = v + 1;
        unlock(mu1);
        lock(mu2);
        v = v + 1;
        unlock(mu2);
}
```

Eraser (Lockset): Key Idea

Every shared variable **must be** guarded by a lock for the **whole computation, AND** the set of locks for each variable is fixed throughout the entire program

- If not, there's a potential race condition

Eraser/Lockset Algorithm

1. Start with the set of all locks that can possibly protect a variable *v (*lockset)
2. If lock *i* is not held when *v* is accesses, remove lock *i* from the lockset of *v*
3. If any of the locksets becomes empty, this indicates a *potential* data race

Eraser Lockset Example

Thread	Program	Locks Held	Candidate Set(v)
1	lock(mu1);	{ }	{mu1, mu2}
1	v = v + 1;	{ mu1 }	
1	unlock(mu1);		{mu1}
2	lock(mu2);	{ }	
2	v = v + 1;	{ mu2 }	{ }
2	unlock(mu2);	{ }	

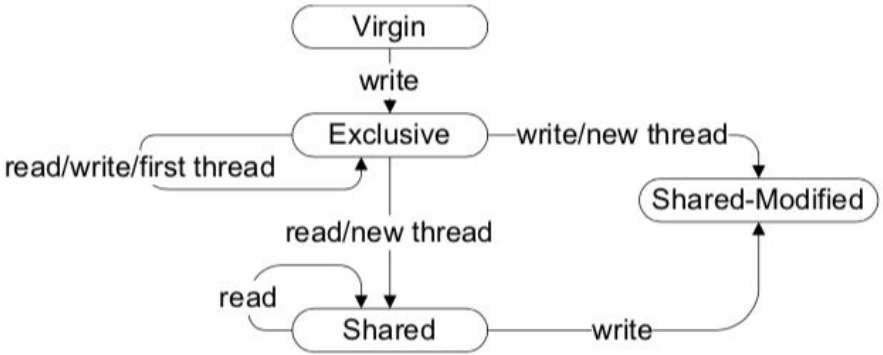
State Machine for Dynamic Lockset Analysis

Simple locking discipline violated by

–initialization of shared variables without holding a lock

–writing shared variables during initialization without locks

–allowing multiple readers in mutual exclusion with single writers



Class Activity

Consider the following code. Apply the state machine to detect if it has a race.

Thread	Program	Locks Held	Candidate Set(v)
1	lock(mu1);		
1	v = v + 1;		
1	unlock(mu1);		
1	y = y + 1		
2	lock(mu2);		
2	v = v + 1;		
2	unlock(mu2);		

Outline

What’s dynamic analysis ?

Pros and Cons of dynamic analysis

Steps in dynamic analysis

Example 1: Memory Tracking

Example 2: Race Detection