Fuzz testing

Tapti Palit

Announcements

- Quiz on Friday on software security
- HW2 questions?

UCDAVIS

What is unit testing?

- Test individual functions
- Add assertions to check if post-conditions hold
- Test corner-cases
- Manually created

What is service tests (integration tests)?

- Test end-to-end functionality
- Manually created

Hard to cover all code with manual testing

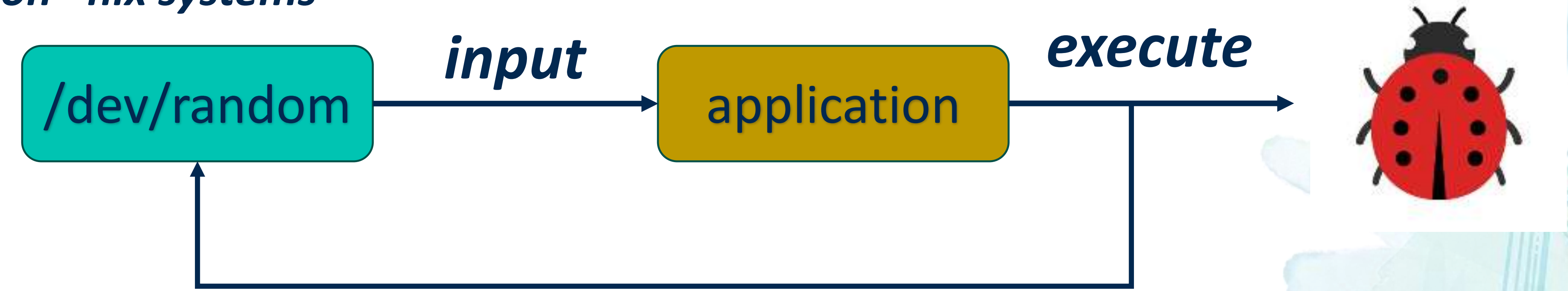
- Linux kernel
  - ~30 million lines of code
- Google's chromium browser
  - ~28 million lines of code
- Hard to manually write test-cases to cover all this code
- ***Need automation***

Fuzz testing

- Goal: to find application inputs that reveal bugs
- Test the entire application instead of individual functions
- Generate inputs randomly until an error is uncovered

Fuzz testing

Source for random input
*on *nix systems*



Black-box testing

Types of bugs

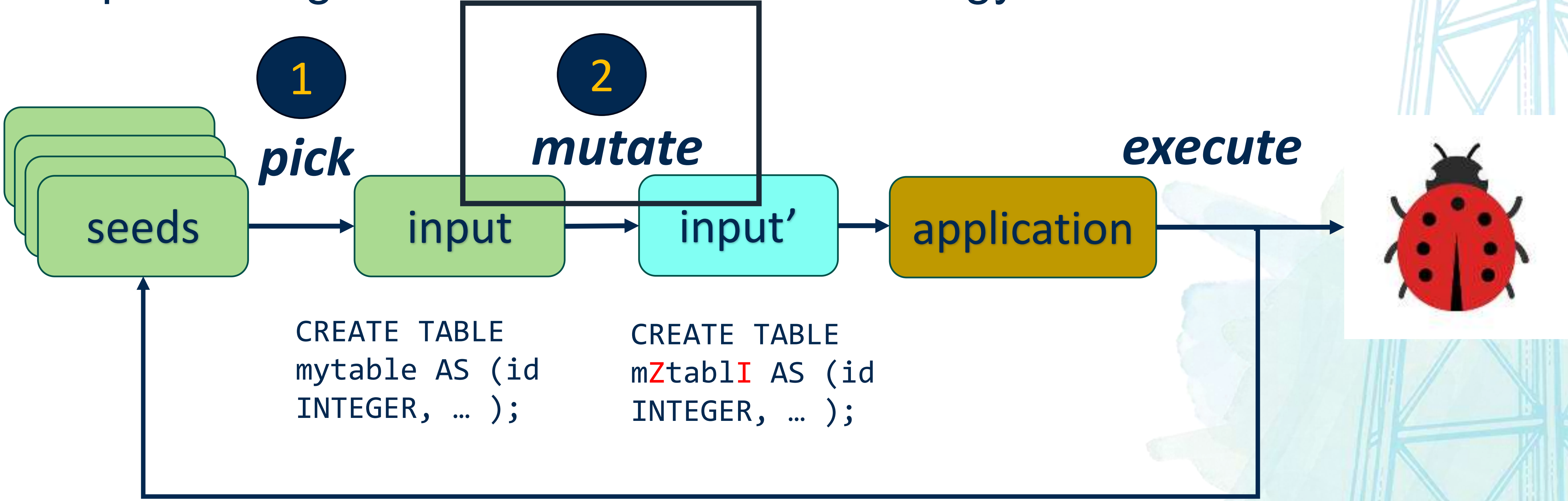
- Incorrect argument validation, incorrect type casting
- Buffer overflows, memory leak, div-by-zero, use-after-free, segmentation fault, and so on…
- Logic errors through assertion failures
- ***Applications written in any language can be fuzzed***

Pros and cons

- Pros: generating random inputs is simple
- Cons: random inputs are not good inputs
- E.g., fuzzing a SQL database software
  - Real inputs: `CREATE TABLE mytable AS (id INTEGER, … );`
  - Random inputs: `aVkjW3txpLZ044zo5kLdUlsU2MtlLNkhwxI8Aew7c0KPfTS115i2rzfBlgod`
  - The database will reject such statements as invalid SQL and not execute the SQL command -> the SQL code will not even be tested

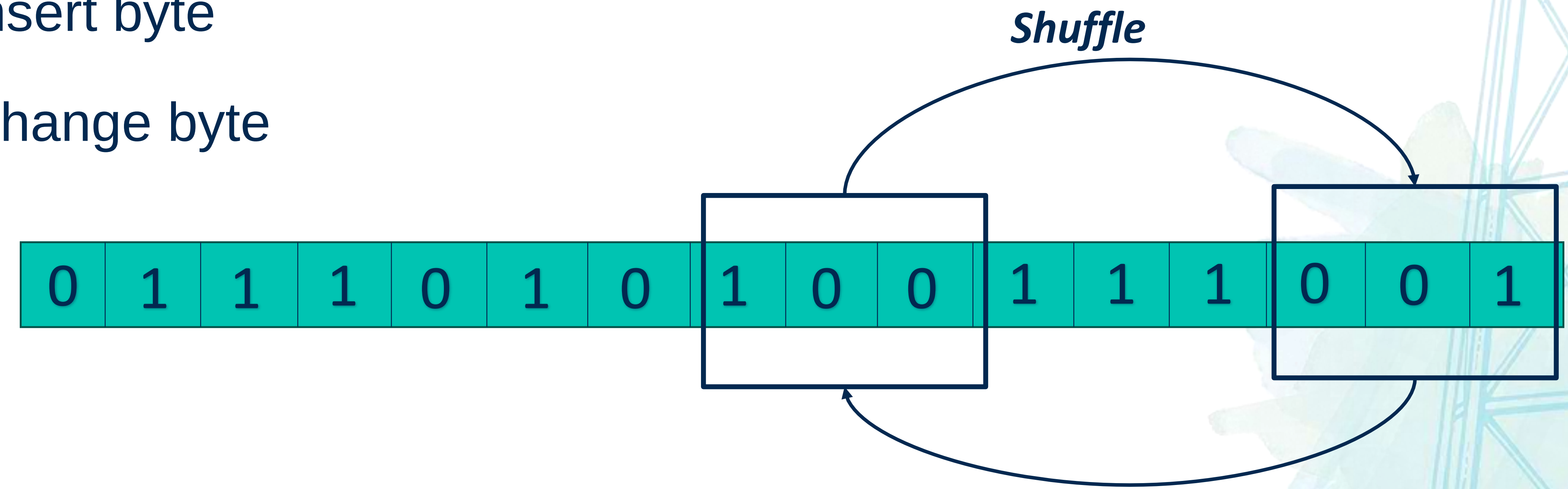
Mutation fuzzer - generate inputs via mutation

- Start with some known valid inputs called *“seeds”*
- Keep mutating them based on some “strategy”



Mutation strategy – byte-level transformations

- Shuffle bytes
- Erase bytes
- Insert byte
- Change byte



Mutation strategy – byte-level transformations

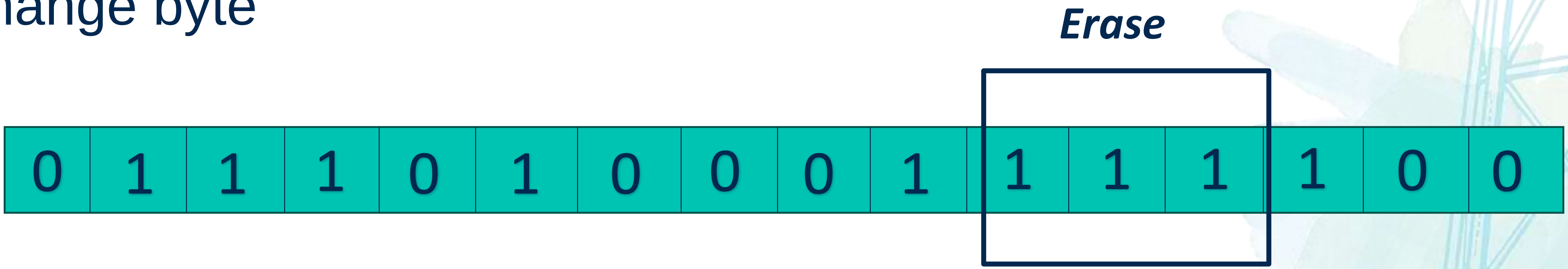
- Shuffle bytes
- Erase bytes
- Insert byte
- Change byte

New input

0	1	1	1	0	1	0	0	0	1	1	1	1	1	0	0

Mutation strategy – byte-level transformations

- Shuffle bytes
- Erase bytes
- Insert byte
- Change byte



Mutation strategy – byte-level transformations

- Shuffle bytes
- Erase bytes
- Insert byte
- Change byte

Next input

0	1	1	1	0	1	0	0	0	1	1	0	0

Mutation strategy – byte-level transformations

- Benefits:
  - Simple to implement
  - Good for binary inputs and streams
    - JPEG file encoder/decoder
- Disadvantages:
  - Generates many invalid inputs for applications that expect structured inputs

Mutation strategy – dictionaries

- Programmer provides a dictionary of "keywords"
  - For SQL, could include `CREATE`, `DELETE`, `INSERT`, `INTO`, `FROM`, `INTEGER`, and so on…
  - For HTML, could include HTML tags `<html>`, `<body>`, `<p>`, and so on…
- Mutate the input by replacing or adding a chosen keyword from this dictionary
- Input - `<html><body><p>Hello world </p></body></html>`
  Dictionary - `{<html>, <body>, <p>,…}`
  Mutated input –
  `<html><body><p> Hello world <p>, Welcome!</p> </p></html>`

UCDAVIS

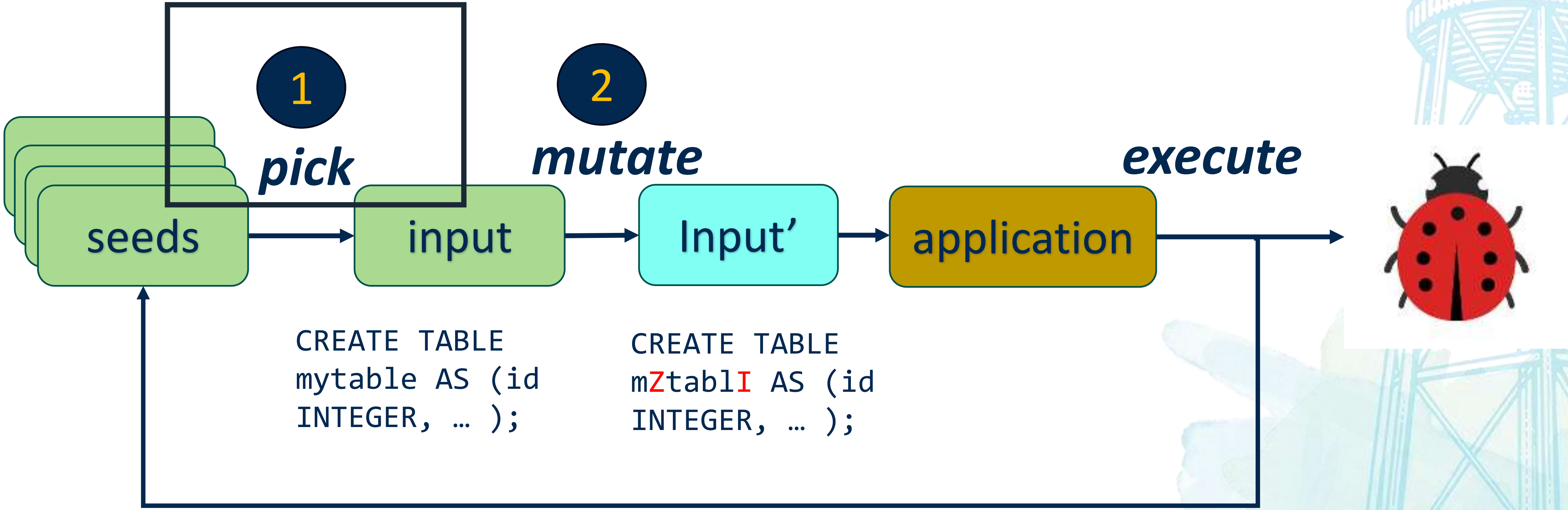
Mutation strategy – dictionaries

- Benefits:
  - Still simple to implement
  - Works reasonably well for structured inputs
- Disadvantages
  - Still generates many invalid inputs for applications that expect structured inputs

Characteristics

- All mutators produce ***many*** invalid inputs
- E.g. malformed HTML, malformed SQL queries
  - `<html><body></P.</ht~l>`
- Still produces enough valid test-cases to be ***very*** effective

Fuzzing pipeline



How to pick which seeds to mutate?

- Goal: Should pick seeds that are “most promising” and likely to find bugs
- What is ***“most promising”?***

Metric of seed "goodness": coverage

- Amount of code or execution paths explored during fuzz testing
- Input seeds that increase coverage are "more promising"
  - Explore new program behaviors
  - Ensures untested paths are tested
- High coverage = high bug-finding potential

Metric of seed “goodness”: coverage

- Amount of code or execution paths explored during fuzz testing

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
            if (b == 10) {
                printf(“Hello!\n”);
            } else {
                printf(“Goodbye!\n”);
            }
        } else {
            printf(“Welcome!\n”);
        }
    } else {
        printf(“Hi!\n”);
    }
}
```

Metric of seed "goodness": coverage

- Amount of code or execution paths explored during fuzz testing
- Input: a = 10, b = 0 -> new coverage!

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
            if (b == 10) {
                printf("Hello!\n");
            } else {
                printf("Goodbye!\n");
            }
        } else {
            printf("Welcome!\n");
        }
    } else {
        printf("Hi!\n");
    }
}
```

Metric of seed “goodness”: coverage

- Amount of code or execution paths explored during fuzz testing
- Input: a = 11, b = 0 -> no new coverage

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
            if (b == 10) {
                printf(“Hello!\n”);
            } else {
                printf(“Goodbye!\n”);
            }
        } else {
            printf(“Welcome!\n”);
        }
    } else {
        printf(“Hi!\n”);
    }
}
```

Metric of seed "goodness": coverage

- Amount of code or execution paths explored during fuzz testing
- Input: a = 10, b = 1 -> new coverage!

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
            if (b == 10) {
                printf("Hello!\n");
            } else {
                printf("Goodbye!\n");
            }
        } else {
            printf("Welcome!\n");
        }
    } else {
        printf("Hi!\n");
    }
}
```

Metric of seed "goodness": coverage

- Amount of code or execution paths explored during fuzz testing
- Input: a = 10, b = 2 -> no new coverage!

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
            if (b == 10) {
                printf("Hello!\n");
            } else {
                printf("Goodbye!\n");
            }
        } else {
            printf("Welcome!\n");
        }
    } else {
        printf("Hi!\n");
    }
}
```

Metric of seed "goodness": coverage

- Amount of code or execution paths explored during fuzz testing
- Input: a = 10, b = 3 -> no new coverage!

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
           if (b == 10) {
                printf("Hello!\n");
           } else {
                printf("Goodbye!\n");
           }
        } else {
           printf("Welcome!\n");
        }
    } else {
        printf("Hi!\n");
    }
}
```

Metric of seed "goodness": coverage

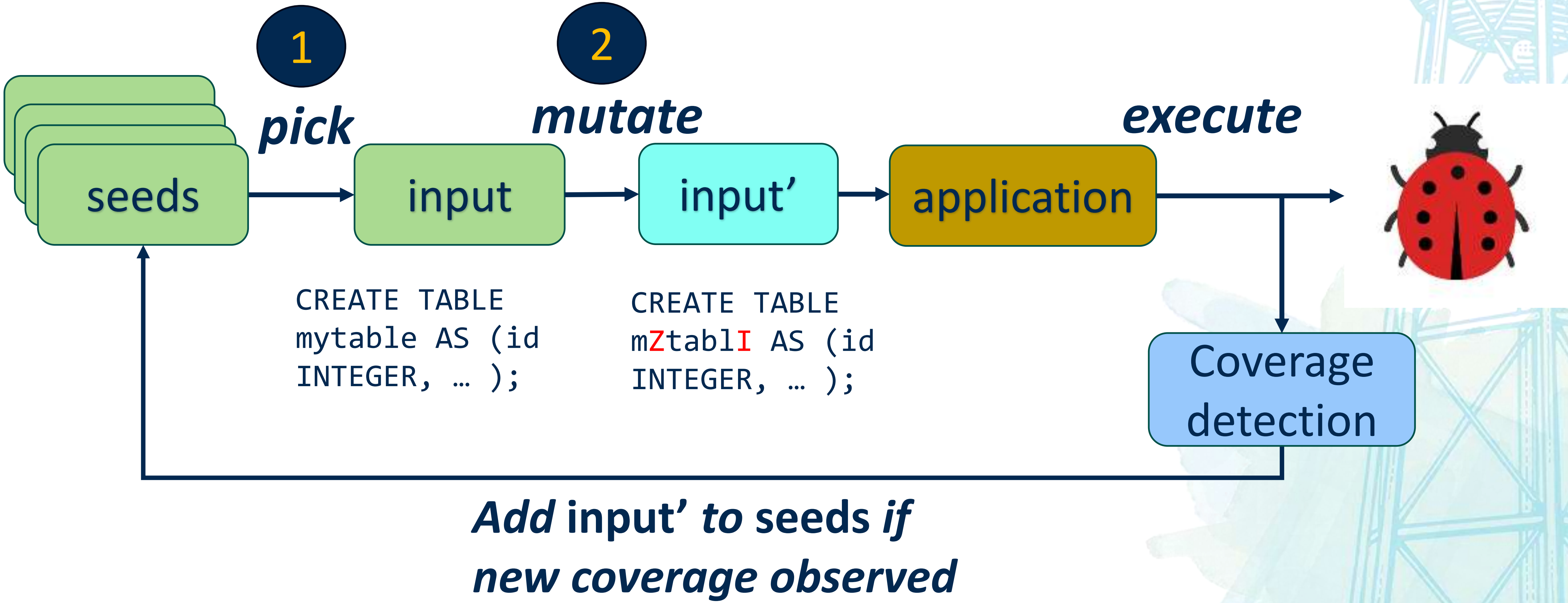
- Amount of code or execution paths explored during fuzz testing
- Input: a = -1, b = 3 -> new coverage!

```
int function(int a, int b) {
    if (a > 0) {
        if (b > 0) {
            if (b == 10) {
                printf("Hello!\n");
            } else {
                printf("Goodbye!\n");
            }
        } else {
            printf("Welcome!\n");
        }
    } else {
        printf("Hi!\n");
    }
}
```

Coverage guided fuzzing

- Record the coverage
- Prioritize mutating input seeds which give higher coverage

Coverage-guided fuzzer pipeline



Fuzzing – high level ideas

- Randomly generate inputs
- Mutation fuzzing
  - Start with known valid inputs
  - Mutate them
    - Byte-level transformations
    - Dictionary-based transformations
- Pick inputs which resulted in new branch coverage for further mutation

AFL demo

- American fuzzy lop (AFL)
- Fuzzer for C/C++ applications

AFL demo

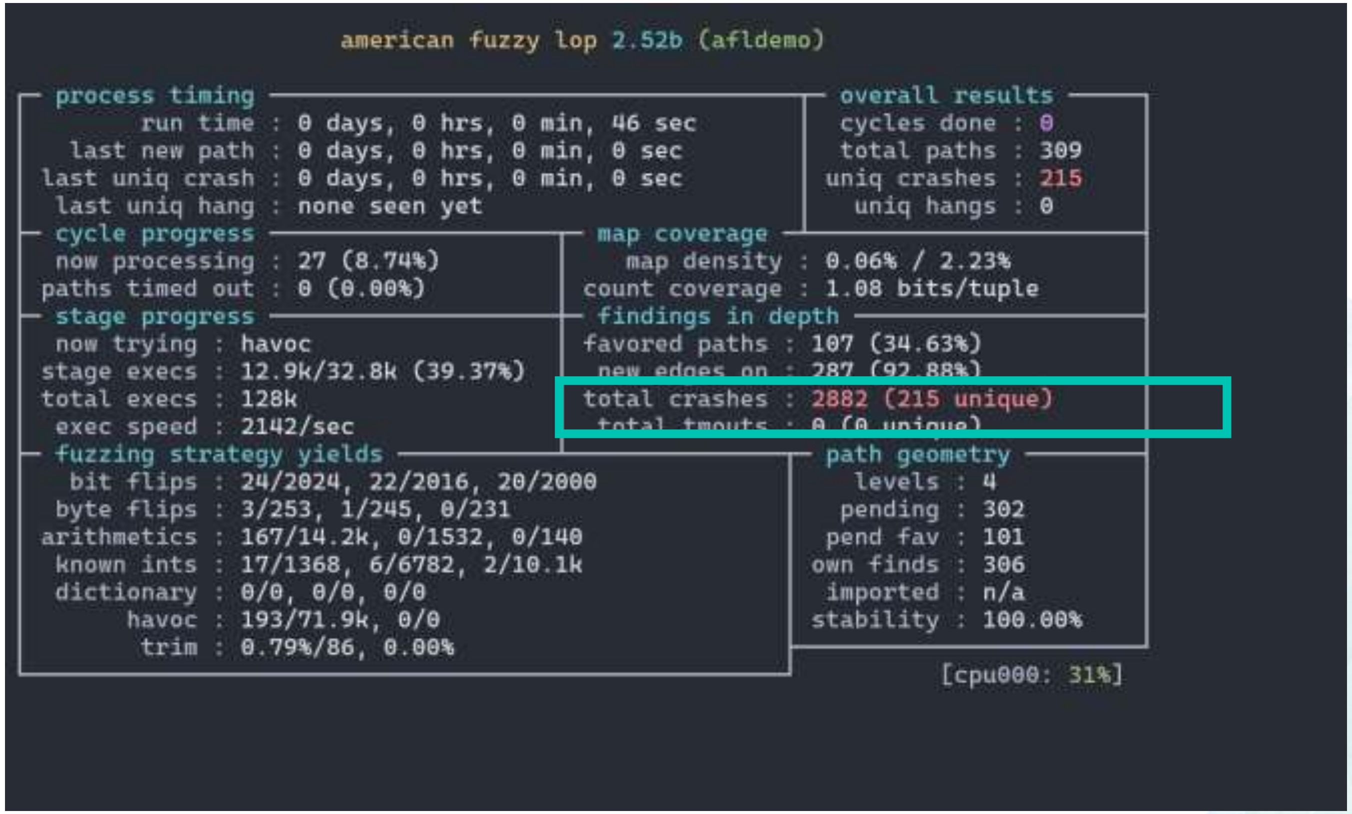
- https://github.com/wolframroesler/afl-demo

AFL demo

- Throughput – number of inputs tested per second
- ***Is higher always better?***

AFL demo

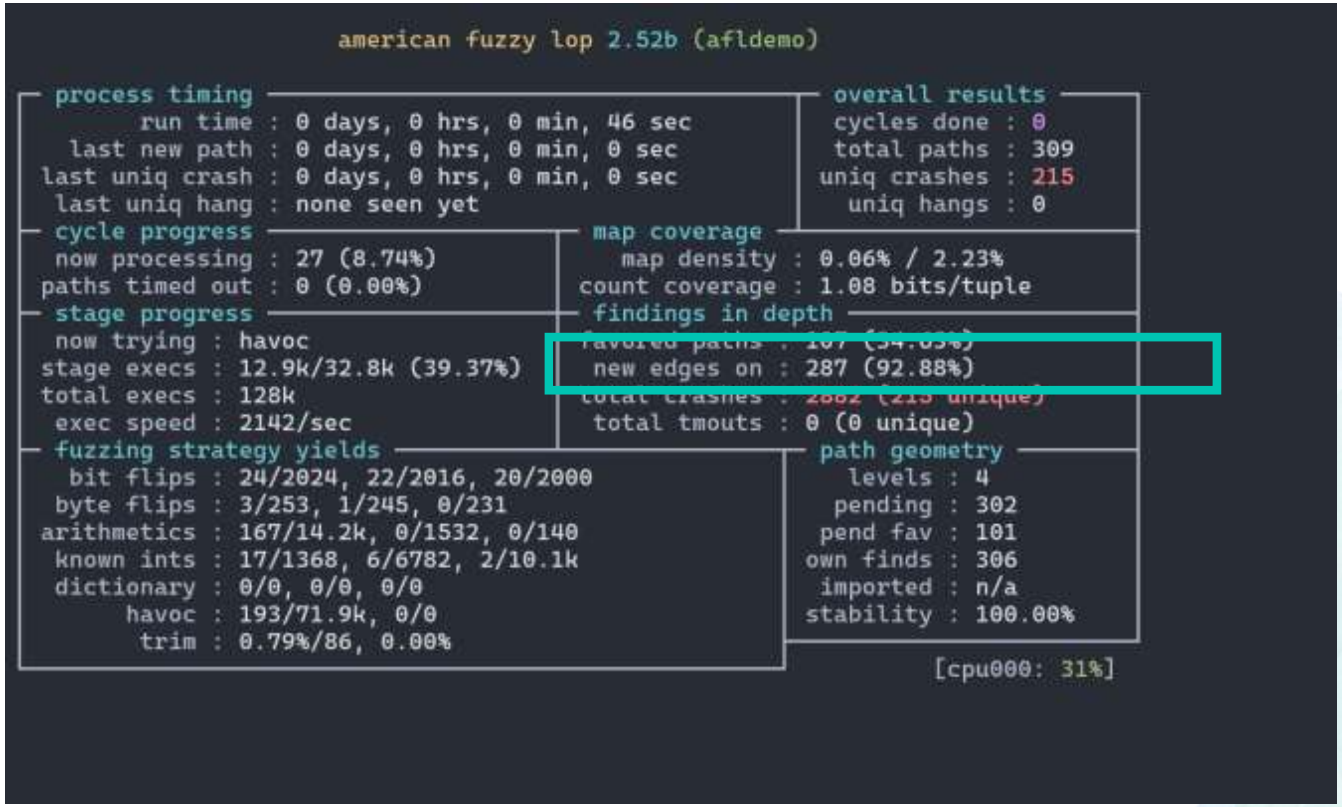
- Crashes found by the fuzzer



Crashes

AFL demo

- Coverage – number of code paths executed



Coverage

Fuzz testing - sanitizers

Tapti Palit

Agenda

- Sanitizers
- Design patterns (intro)
- Quiz (15 minutes/ 30 minutes)

Recall: memory safety bugs

- Fuzzers execute the program with random inputs and look for crashes
- But memory safety bugs may or may not crash the program
- **Undefined Behavior (UB)**

Sanitizers: oracles for memory safety bugs

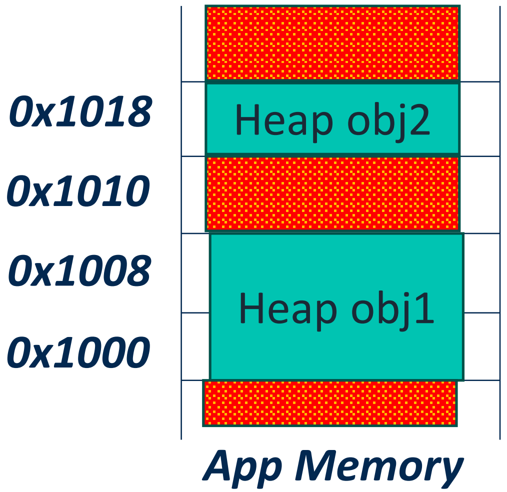
- Adds additional compiler instrumentation to record object metadata
- Turns silent UBs to crashes
- Popular sanitizers
  - Address sanitizer (ASAN) – can detect buffer overflow
  - Leak sanitizer (LSAN) – can detect memory leak
  - Thread sanitizer (TSAN) – can detect data races
- ***Significantly*** slows down the application; but can detect memory safety bugs!!

Address sanitizer

- Goal – detect buffer overflow and UAF
- Insert “red/poison zones” around memory
- Updates to red zones indicate buffer overflow
  - Need to detect writes to red zones

```
void function(char* p) {
    while(*p != ‘\0’) *p++ = ‘a’;
}

int main(void) {
    char* s = malloc(16); // heap object 1
    char* p = malloc(8); // heap object 2
    function(s);
}
```

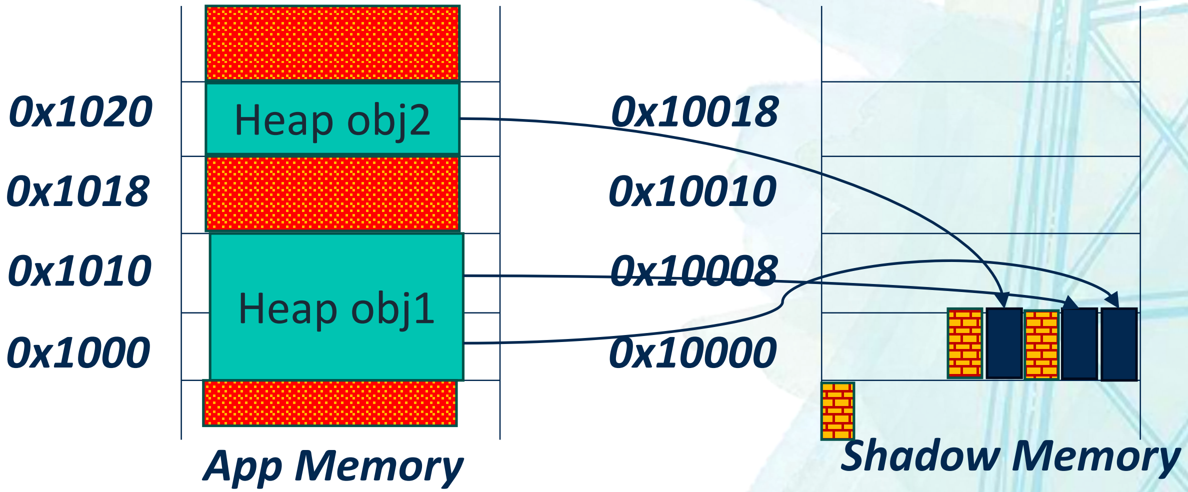


Shadow memory

- Maintains shadow memory to record metadata about red and accessible zones
- Each ***byte*** in shadow memory represents a corresponding ***8 bytes*** in app
  - 0x00 for valid
  - 0xFF for poisoned range
  - 0x01 – 0x07 – partially poisoned

```
void function(char* p) {
     while(*p != '\0') *p++ = 'a';
}


int main(void) {
    char* s = malloc(16); // heap object 1
    char* p = malloc(8); // heap object 2
    function(s);
}
```

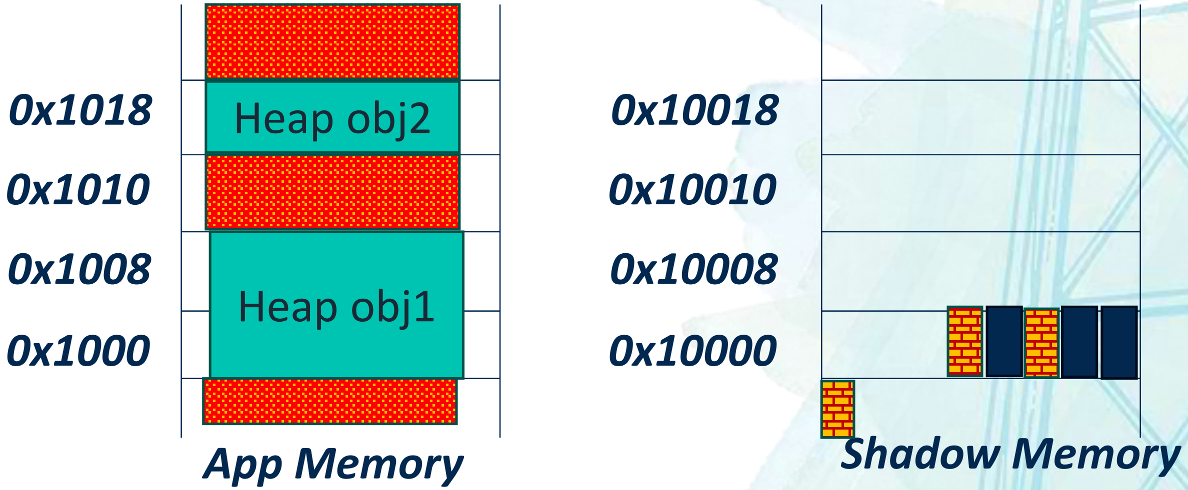


Compiler checks

- Compiler adds instructions before each memory access to first check the shadow memory byte

```
void function(char* p) {
     while(*p != '\0') *p++ = 'a';
}


int main(void) {
    char* s = malloc(16); // heap object 1
    char* p = malloc(8); // heap object 2
    function(s);
}
```

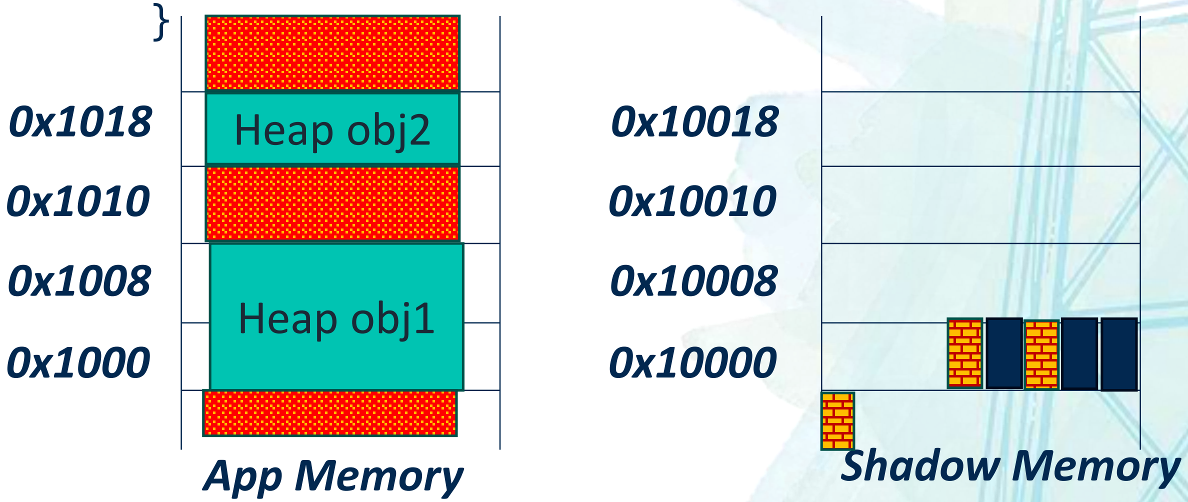


Compiler checks pseudocode

- Compiler adds instructions before each memory access to first check the shadow memory byte
- Pseudocode
  - shdw_poisoned(void*) computes the address of the shadow byte
  - Checks if it is fully/partially poisoned

```
void function(char* p) {
     while(*p != '\0') {
         if (shdw_poisoned(p)) { assert(false); }
         *p++ = 'a';
     }
}

int main(void) {
     char* s = malloc(16); // heap object 1
     char* p = malloc(8); // heap object 2
     function(s);
}
```



Why shadow memory?

- Shadow memory stores metadata about the 8-byte memory range
- Fast computation of shadow byte address
  - `shadow = (address >> 3) + SHADOW_OFFSET`
- Fast checks of shadow byte value
  - `shadow_value = *shadow`

Summary

- Fuzzing is an effective testing mechanism
- Unlike other testing approaches it can ***find new inputs*** that cause program misbehavior
- Mutation and coverage are important for generating effective fuzzers
- Sanitizers provide additional information needed to find *silent UB* bugs

UCDAVIS