POLITECNICO
MILANO 1863

Software Engineering 2

Automated Testing

Concolic Execution

Fuzzing

M Camilli, E Di Nitto, M Rossi

Verification & Validation

Automated Testing: Concolic (Concrete-Symbolic) Execution

Symbolic execution: test case generation

POLITECNICO
MILANO 1863

- Symbolic execution can be used to test specific execution paths automatically
  - Give as input a target location or some paths
  - If the path conditions are SAT
    - Generate N satisfying assignments (i.e., test cases)
    - Execute the test cases and, for each execution, check the reached state (with an oracle)
- However, we already discussed the **weaknesses** of symbolic execution…

Concrete-symbolic (concolic) execution

- The very idea
  - Perform symbolic execution **alongside** a concrete one (concrete inputs)
  - The **state** of concolic execution combines a **symbolic** part and a **concrete** part, used as needed to make progress in the exploration
- Steps
  - Concrete to symbolic: derive conditions to explore new paths
  - Symbolic to concrete: simplify conditions to generate concrete inputs

Concolic execution: example
(concrete to symbolic)

- **Example**: Let’s explore all paths of procedure `m`
- We start from a (random) concrete input, at the same time, we build the symbolic condition of the explored path

```
0: void m(int x, int y) {
1:   int z := 2 * y
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print(“Log message.”)
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
22	7		

<0,1>

x	y	z	π
X	Y	2Y	T
22	7	14	

<0,1,2>

x	y	z	π
X	Y	2Y	2Y≠X
22	7	14	

<0,1,2,6,7>

x	y	z	π
X	Y	2Y	2Y≠X
22	7	14	

- {x=22, y=7} → path `<0,1,2,6,7>`, path condition: 2Y≠X
- To explore another path, negate the path condition: ¬(2Y≠X)

Concolic execution: example
(symbolic to concrete)

x	y	z	π
X	Y		T
2	1		

<0>

- If we can solve the constraint, we start again with another concrete input that satisfies the new constraint ¬(2Y≠X)

```
void m(int x, int y) {
  int z := 2 * y
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: ¬(2Y≠X)

Concolic execution: example
(concrete to symbolic)

x	y	z	π
X	Y		T
2	1		

<0>

x	y	z	π
X	Y	2Y	T
2	1	2	

<0,1>

- We explore the new path and apply again the concrete-to-symbolic step

```
0: void m(int x, int y) {
1:   int z := 2 * y
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

x	y	z	π
X	Y	2Y	2Y=X
2	1	2	

<0,1,2>

x	y	z	π
X	Y	Y+10	2Y=X
2	1	11	

<0,1,2,3>

- {x=2, y=1} → path <0,1,2,3,4,5,6,7> with path condition $2Y=X \land X \leq Y+10$

<0,1,2,3,4,5,6,7>

x	y	z	π
X	Y	Y+10	2Y=X
2	1	11	X≤Y+10

Partial negation: last condition only

$2Y=X \land \neg(X \leq Y+10)$

Concolic execution: example
(symbolic to concrete)

x	y	z	π
X	Y		T
30	15		

- New input values are identified

```
void m(int x, int y) {
  int z := 2 * y
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: $2Y=X \land \neg(X \leq Y + 10) \equiv$
$2Y=X \land X > Y + 10$

Concolic execution: example
(concrete to symbolic)

- We explore the new path and apply again the concrete-to-symbolic step

```
0: void m(int x, int y) {
1:   int z := 2 * y
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
30	15		

<0,1>

x	y	z	π
X	Y	2Y	T
30	15	30	

<0,1,2>

x	y	z	π
X	Y	2Y	2Y=X
30	15	30	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	2Y=X
30	15	25	

<0,1,2,3,4,6,7>

x	y	z	π
X	Y	Y+10	2Y=X
30	15	25	X>Y+10

- Conclusion: we have been able to cover all paths with the following test cases:
  - <0,1,2,6,7>: {x=22, y=7}
  - <0,1,2,3,4,5,6,7>: {x=2, y=1}
  - <0,1,2,3,4,6,7>: {x=30, y=15}

Concolic execution: example2

- **Example**: Let's explore again all paths of the procedure `m2`

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

- We try to follow the same approach, but, in some cases, we cannot solve the symbolic condition…
- Behavior of `bb` is unknown. We execute it with the identified input cases
  - Example: run bb(7) returns 14
- Now the condition can be solved

x	y	z	π
X	Y		T
22	7		

<0>

x	y	z	π
X	Y	bb(Y)	T
22	7	14	

<0,1>

<0,1,2,6,7>

x	y	z	π
X	Y	bb(Y)	bb(Y)≠X
22	7	14	

Negate: ¬(bb(Y)≠X)

Solve: ¬(bb(Y)≠X) → cannot do it symbolically!

Symbolic-to-concrete step

¬(14≠X) ≡ 14=X

Concolic execution: example2

x	y	z	π
X	Y		T
14	7		

<0>

- Now the constraint can be solved and we can start a new exploration

```
void m2(int x, int y) {
  int z := bb(y) //black-box function
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: $\neg(14 \neq X) \equiv 14 = X$

Concolic execution: example2

- New explorations follow the same approach

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
14	7		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
14	7	14	

<0,1,2>

x	y	z	π
X	Y	bb(Y)	bb(Y)=X
14	7	14	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	bb(Y)=X
14	7	17	

<0,1,2,3,4,5,6,7>

x	y	z	π
X	Y	Y+10	bb(Y)=X
14	7	17	X≤Y+10

Negate last condition:
bb(Y)=X ∧ ¬(X≤Y+10) ≡ bb(Y)=X ∧ X>Y+10

Solve: bb(Y)=X ∧ X>Y+10 → cannot!

Concolic execution: example2

x	y	z	π
X	Y		T
34	17		

<0>

POLITECNICO MILANO 1863

- New explorations follows the same approach

```
void m2(int x, int y) {
  int z := bb(y) //black-box function
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Concretize: bb(Y)=X ∧ X>Y+10
We select Y randomly, execute bb(Y) and check if the formula holds
Example: Y=17, bb(Y)=34
Solve: Y=17 ∧ bb(Y)=34 ∧ bb(Y) =X ∧ X>Y+10

Concolic execution: example2

POLITECNICO MILANO 1863

- New explorations follows the same approach

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
34	17		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
34	17	34	

<0,1,2>

x	y	z	π
X	Y	bb(Y)	bb(Y)=X
34	17	34	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	bb(Y)=X
34	17	27	

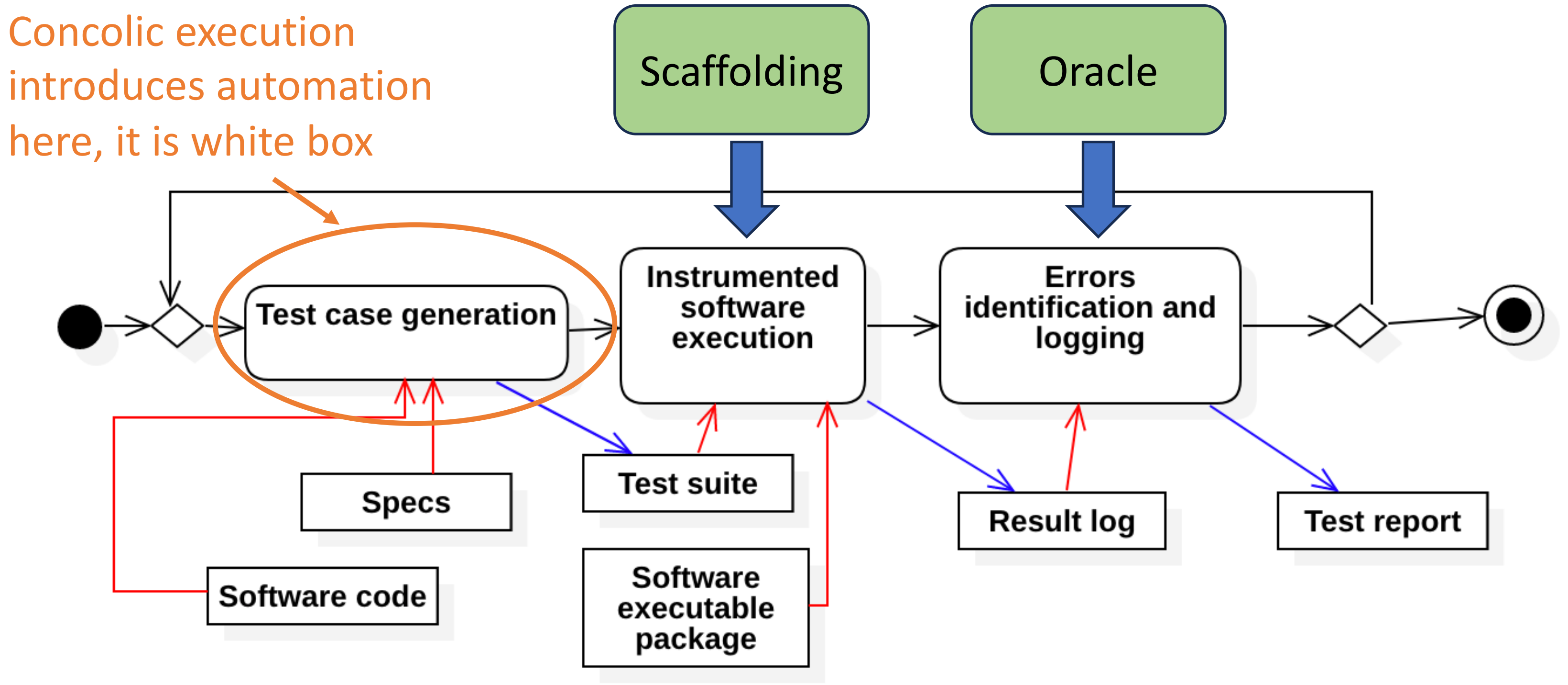
<0,1,2,3,4,6,7>

x	y	z	π
X	Y	Y+10	bb(Y)=X
34	17	27	X>Y+10

Concolic execution: pros and cons

- Advantages
  - Can deal with black-box functions in path conditions (not possible with symbolic exec)
  - Can generate concrete test cases automatically, according to some code coverage criterion
- Limitations
  - Finds just **one input example** per path, however…
    - Faults typically occur with certain inputs only
    - If faults are rare events, it's unlikely to spot them with concolic exec
  - #paths explode due to complex nested conditions → **large search space**
  - **Does not guide the exploration**, it just explores possible paths one by one as long as we have budget (e.g., time, #runs)

Positioning concolic execution in the testing workflow



Verification & Validation

Automated Testing: Fuzz Testing (fuzzing)

Fuzzing, motivations

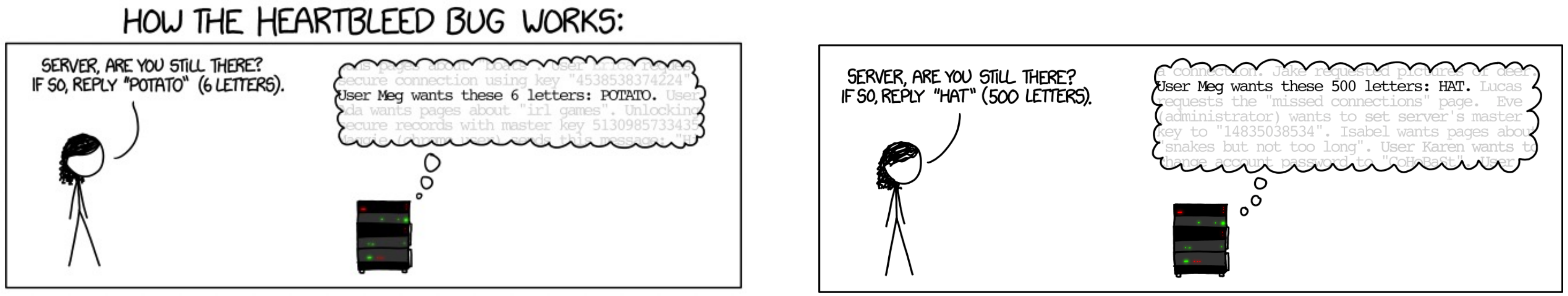
POLITECNICO
MILANO 1863

- Complements functional testing (e.g., manually defined test cases based on functional requirement specs)
  - Works at **component** or **system** level
  - Deals also with external qualities other than correctness, e.g., **reliability** and **security** by providing **randomly** generated data as inputs
  - Can uncover defects that might not be caught by other methods since **randomness** typically leads to **unexpected**, or **invalid** inputs
- Essence of fuzzing
  - Create random inputs, and see if they break things
  - Let it run long enough, and you'll see

HeartBleed bug, a famous example

https://heartbleed.com/

- Security bug in OpenSSL library (cryptographic protocols for remote communication)
- Unchecked memory access
  - Exploited by sending a specially crafted command to the SSL heartbeat service
  - Introduced in 2012 → discovered and fixed in 2014
  - Detected by researchers at Google using fuzzing + runtime memory-checks



Continue…
Full strip: https://xkcd.com/1354/

Fuzzing, an example

- Let's assume we want to **fuzz an existing program**, such as `bc`
  - `bc` is a UNIX utility "basic calculator"
  - It expects as input a stream of chars representing mathematical expressions
    - Example: `echo "(1+3)*2" | bc` returns `8`
- We evaluate the robustness of `bc` given an unpredictable input stream following these steps:
  - **Build a fuzzer**: a program that will output a random char stream
  - **Attack** `bc` using the fuzzer with the goal of trying to break it

A simple fuzzer

- Let's build a simple fuzzer (fuzz generator)..
  - We use Python (you can see it as pseudocode if it is not familiar)

```python
# A string of up to `max_length` characters in the range
#  [`char_start`, `char_start` + `char_range`)

def fuzzer(max_length: int = 100, char_start: int = 32, char_range: int = 32) -> str:
    string_length = random.randrange(0, max_length + 1)
    out = ""
    for i in range(0, string_length):
        out += chr(random.randrange(char_start, char_start + char_range))
    return out
```

random integer in [0, max_length]

chr(*n*) returns the character with ASCII code *n*

A simple fuzzer

POLITECNICO MILANO 1863

- What is a possible output of `fuzzer()` with default args?

max_length = 100

char_start = 32

'!7#%"*#0=)$;%6*;>638:*>80"=</>(/*:-(2<4 !:5*6856&?""11<7+%<%7,4.8,*+&,,$,."'

char_start + char_range = 64

ASCII table extract

Decimal	Hex	Char
32	20	[SPACE]
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
...		
64	40	@
65	41	A

A simple fuzzer

- Fuzz input = term for such random, unstructured data
- Assume a program expects a specific input format (e.g., comma-separated list of values, e-mail address)
  - What happens if we give fuzz strings as input?
  - Can the program process such an input without any problems?
- Note: fuzzers can produce any kinds of input, for example:
  - sequence of lowercase letters (up to 100)
  - up to 1000 random digits
  - up to 200 alternating lowercase and uppercase letters
  - Other data types: integer, float, …
  - **Exercise**: use and/or modify `fuzzer()` to produce these fuzz inputs

Fuzzing external programs

POLITECNICO
MILANO 1863

- To test the program we create an input file (test data); then we feed this input file to `bc`
- Here is an example of manually defined test case

```
import os
import tempfile, subprocess

basename = "input.txt"
tempdir = tempfile.mkdtemp()
FILE = os.path.join(tempdir, basename) # tmp file s.t. we do not clutter the file system

program = "bc" # simple unix calc utility
input = "2 + 2\n" # nominal input
with open(FILE, "w") as f:
    f.write(input)
result = subprocess.run([program, FILE],
                        stdin=subprocess.DEVNULL,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        universal_newlines=True)
```

Note:
executes an external program, similar to
`Runtime.getRuntime().exec(…)` in Java
`system(…)` in C

Fuzzing external programs

- Let's now feed a large number of inputs to `bc`, to see whether it might crash on some
- **Note**: running this may take a while.. so, we need to define a reasonable **budget**

```
trials = 100 # testing budget
program = "bc"

runs = []
for i in range(trials):
    with open(FILE, "w") as f:
        data = fuzzer()
        f.write(data)
    result = subprocess.run([program, FILE],
                            stdin=subprocess.DEVNULL,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            universal_newlines=True)
    runs.append((data, result))
```

As long as we have budget

At the end runs contains all generated inputs with corresponding results

Fuzzing external programs

- From the result, we can check the program output and the status

```
result.stdout # e.g., '4\n'
result.returncode # e.g., 0 is terminated correctly
result.stderr # e.g., '' if no error msgs
```

Fuzzing external programs — Warning

- Note: instead of `bc`, you can put in any program (e.g., any UNIX utility)
- Be aware that if the program can change your system, fuzz inputs **may cause damages**!
  - **Example**: let's imagine we test: `rm -fr FILE`
  - **Exercise**: What is the chance of causing damages?
    - e.g., "`/<white space><other>`" will erase your entire file system
      e.g., "`~<white space><other>`" will erase your home dir
      e.g., "`.<white space><other>`" will erase the current folder

Fuzzing external programs — Result log

- After the execution, we can query the `runs` structure (i.e., our result log)
  - How many runs passed? (no error messages)

    ```
    sum(1 for (data, result) in runs if result.stderr == "")
    ```

  - how many crashed?

    ```
    sum(1 for (data, result) in runs if result.returncode != 0)
    ```

  - We can also inspect possible error messages:

    Parse error: bad character '&' /var/folders/…/input.txt:1

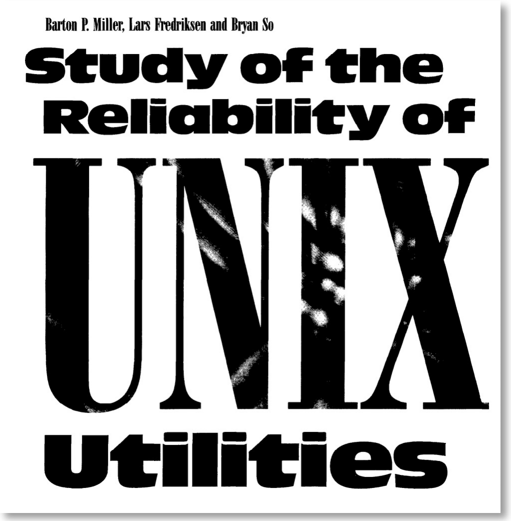
    Not very interesting…
  - Any runs with messages other than illegal character, parse error, or syntax error? Try out and see…

    ```
    [result.stderr for (data, result) in runs if result.stderr != ""
      and "illegal character" not in result.stderr
      and "parse error" not in result.stderr
      and "syntax error" not in result.stderr]
    ```

Fuzzing external programs

- [Miller et al., 1990] About a third of the UNIX utilities (including the *bc* utility) they fuzzed had issues – crash, or hang when tested with fuzz inputs
- Apparently, the bugs have been fixed.. you can try to replicate the experiment!

Barton Miller et al., 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33, 12 (Dec. 1990), 32–44.
https://doi.org/10.1145/96267.96279
Assignment Miller gave to his students to build the research:
https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf



Common bugs found by fuzzers

- Buffer overflow
  - Many programs have built-in maximum lengths for inputs
  - In languages like *C*, it is easy to exceed these lengths, thus causing buffer overflows

```
char weekday[9];  // 8 characters + trailing '\0' terminator
…
strcpy(weekday, input);
```

What if `input` is "Wednesday"?

"\0" is copied in memory after `weekday`
→ it may overwrite other vars causing arbitrary behavior

Common bugs found by fuzzers

- Missing error check
  - Many languages, like *C*, do not have exceptions, instead they have functions that return special error codes in exceptional circumstances:

```
/* getchar returns a character from stdin; if no input is available
anymore, it returns EOF */

while (getchar() != ' ');
```

Here the program reads the input, char by char until a whitespace (`' '`) occurs… what if the input ends (`EOF`) before a whitespace?

`getchar()` returns `EOF`, and keeps on returning `EOF` when called again
→ the code enters an infinite loop (hangs)

Common bugs found by fuzzers

- Rogue numbers
  - Fuzzing easily generates uncommon values in the input, causing interesting behavior

```
/* reads size from the input, and then allocates a buffer of the given size */

char* read_input() {
    size_t size = read_size();
    char *buffer = (char *)malloc(size);
    // fill buffer
    return buffer;
}
```

What if `size` is very large, exceeding program memory? What if `size` is less than the number of characters in `stdin`? What if `size` is negative?

→ by providing a random number for `size`, fuzzing can create all kinds of damages

Checking memory accesses

- Best practice: fuzzing + runtime memory-checks
  - Catch problematic memory accesses during testing by instrumenting programs with memory-checking environments
    - Example: *C* lang Address Sanitizer https://clang.llvm.org/docs/AddressSanitizer.html
  - Instrumentation checks at runtime every memory operation to detect potential violations
    - Out-of-bounds accesses to heap, stack
    - Use-after-free
    - Double-free
  - Cost-effectiveness trade-off
    - Increases execution time (typical slowdown is 2x and more memory)
    - Decreases human effort to find these bugs

Fuzzing with mutations: problem

- Problem
  - Many programs expect inputs in very specific formats before they would actually process them
  - In this case, completely random inputs have low chance to execute deep paths
- Example
  - Assume we have a program that accepts a URL
  - What is the chance of producing a valid URL when fuzzing with random inputs?

Fuzzing with mutations: problem

- More details about the example...
- A URL has a number of elements: `scheme://netloc/path?query#fragment`
    - `scheme` is the protocol (http, https, ...)
    - `netloc` is the host name (e.g., www.google.com)
    - `path` is the path on that host (e.g., search)
    - `query` is a list of key-value pairs (e.g., q=fuzzing)
    - `fragment` is a marker for a location in the retrieved document (e.g., #result)
- Let's use `fuzzer(char_start=32, char_range=96)`
    - we use the full range of printable ASCII characters (including `:` and `/`)
    - we need a string starting with "http://" or "https://"

Fuzzing with mutations: problem

- Let’s use `fuzzer(char_start=32, char_range=96)`
  - we need a string starting with "http://" or "https://"
- What is the chance?
  - We have two sequences of 7 and 8 very specific characters
    - $(1/96)^7 + (1/96)^8 = 1.3446 * 10^{-14}$ (likelihood)
- What is the time required to produce a valid URL?
  - Assume a single run is very fast, say, ~1 microsec?
    - 1/likelihood = $7.4370 * 10^{13}$ (avg #runs)
    - 10^{-6} * avg #runs = $7.430 * 10^7$ seconds ≈ 20658 hrs ≈ 860 days ≈ 2.4 years

Fuzzing with mutations: the idea

- In this case we wait 2.4 years on average to get a single valid URL
  - → good chance of finding bugs in input parsing
  - → little chance of reaching any deeper functionality
- The generation should be guided!
- Mutational fuzzing: rather than random inputs from scratch (generational fuzzing), we mutate a given valid input
  - Mutation = simple input manipulation
    - e.g., with strings we can insert a character, delete a character, or flip a bit in character representation

Fuzzing with mutations: example

```
import random

def insert_random_char(s: str) -> str:
    pos = random.randint(0, len(s))
    random_char = chr(random.randrange(32, 127)) // rand printable char
    return s[:pos] + random_char + s[pos:]
…

def mutate(s: str) -> str:
    mutators = [ delete_random_char,
                 insert_random_char,
                 flip_random_char ]
    mutator = random.choice(mutators)
    return mutator(s)
```

Implementation of mutation operators (mutators)

Generates a fuzz input by applying a random mutator

Fuzzing with mutations: example

- We can even apply multiple mutations, e.g., 20 or 50 mutations

```
seed_input = "http://www.google.com/search?q=fuzzing"
mutations = 50
fuzz_input = seed_input
for i in range(mutations):
    print("{} mutations: {}".format(i, fuzz_input))
    fuzz_input = mutate(fuzz_input)
```

```
0 mutations: http://www.google.com/search?q=fuzzing
…
10 mutations: http:/L/www.ggoWglej.com/seaRchqfu:in
…
30 mutations: htv://>fwggoVgle"j.qom/ea0Rd3hqf,u^:i
…
50 mutations: htv://>fwgeo]6zTle"BjM.\'qom/eaR[3hqf,tu^:i
```

Observations

- Multiple mutations
  → higher variety of inputs
- Too many mutations
  → higher chance of invalid inputs

Fuzzing: guiding by coverage

- The higher the variety → the higher the risk of an invalid input
- We should keep and mutate inputs that are especially valuable
  - Guiding by **coverage** information is a popular approach
    - Coverage (line, branch, or path) is a common metric for test quality
    - In this case we say the test case generation is **gray box**

- The very idea
  - Evolve only those test cases that have been successful
    - Success = found a new path
  - Fuzzer keeps and maintains a population of successful inputs; if a new input finds another path, it will be retained as well

Fuzzing: guiding by coverage

- We define a utility function to run test cases
  - We assume there exists a `Coverage` class that we can use to retrieve coverage of a test run

```
PASS = "PASS"
FAIL = "FAIL"

def run_function(foo: Callable, inp: str) -> Any:
    with Coverage() as cov:
        try: result = foo(inp)
             outcome = PASS
        except Exception:
            result = None
            outcome = FAIL
    return result, outcome, cov.coverage()
```

The `coverage()` method returns the coverage achieved in the last run as list of tuples:
<function-name, executed-line number>

[('urlsplit', 465),
('urlparse', 394),
('urlparse', 400)]

#lines executed in the last run

Fuzzing: guiding by coverage

```
def mutation_coverage_fuzzer(foo: Callable, seed: List[str], min_muts: int = 2,
        max_muts: int = 10, budget: int = 100) -> List[Tuple[int, str, int]]:
    population = seed
    cov_seen = set()
    summary = []
    for j in range(budget):
        candidate = random.choice(population)
        trials = random.randint(min_muts, max_muts)
        for i in range(trials):
            candidate = mutate(candidate)
        result,outcome,new_cov = run_function(foo, candidate)
        if outcome == PASS and not set(new_cov).issubset(cov_seen):
            population.append(candidate)
            cov_seen.update(new_cov)
            summary.append((j, candidate, len(cov_seen)))
    return summary
```

create candidate test case

run test case with coverage

evaluate “goodness” of test case

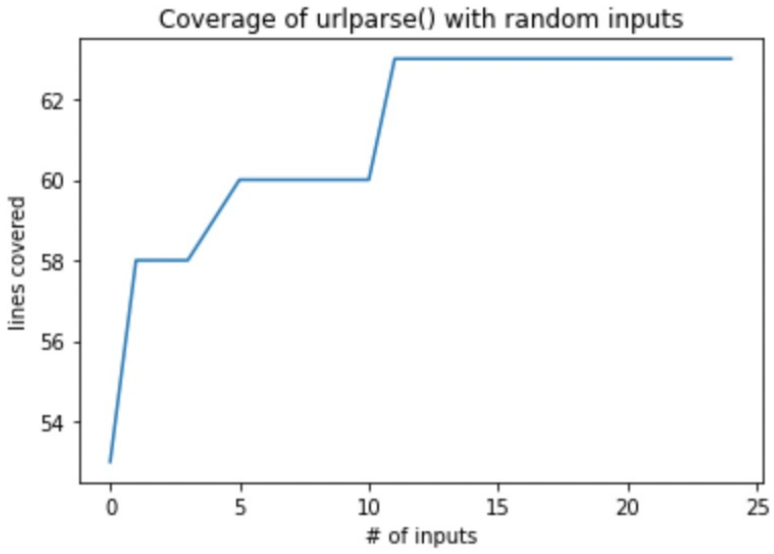
Set union

Fuzzing: guiding by coverage

- Let's test a target program `urlparse` with 10k fuzz inputs

```
seed_input = "http://www.google.com/search?q=fuzzing"
summary = mutation_coverage_fuzzer(urlparse, [seed_input], budget = 10000)
```

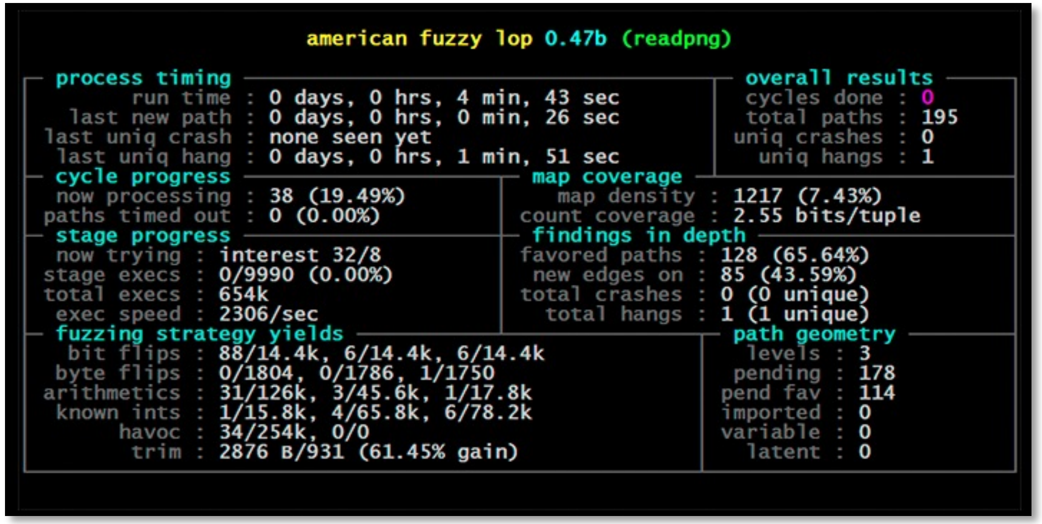
- By inspecting `summary` we can see each and every input is valid and has different coverage!
- Strengths
  - Practical also with large programs – it explores one path after the other until you have budget
  - All you need is a way to capture the coverage



AFL, a succesful story

- [Nov 2013] 1st version of American Fuzzy Lop (AFL) was released, one of the most successful fuzzing tools
  - https://lcamtuf.coredump.cx/afl/

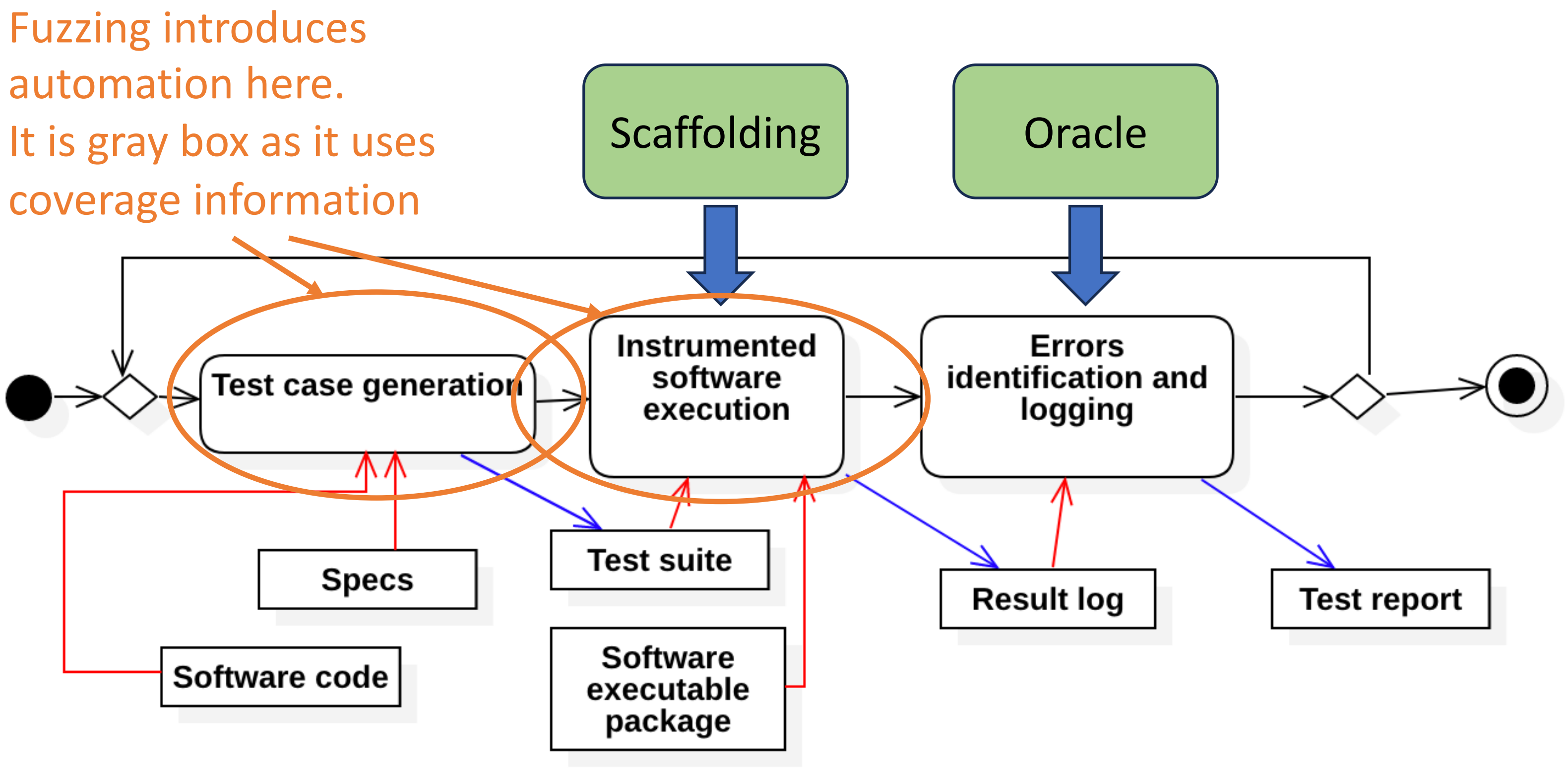
- First to demonstrate that failures and vulnerabilities can be detected automatically in large-scale security-critical applications
- Implements mutational fuzzing + guiding by coverage



AFL User Interface

Positioning fuzzing in the testing workflow

POLITECNICO
MILANO 1863



References

- Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler: "The Fuzzing Book". Retrieved 2023-11. https://www.fuzzingbook.org/