POLITECNICO
MILANO 1863

Software Engineering 2

Fuzzing

Search-based Software Testing

M Camilli, E Di Nitto, M Rossi

Verification & Validation

Automated Testing: Fuzz Testing (fuzzing)

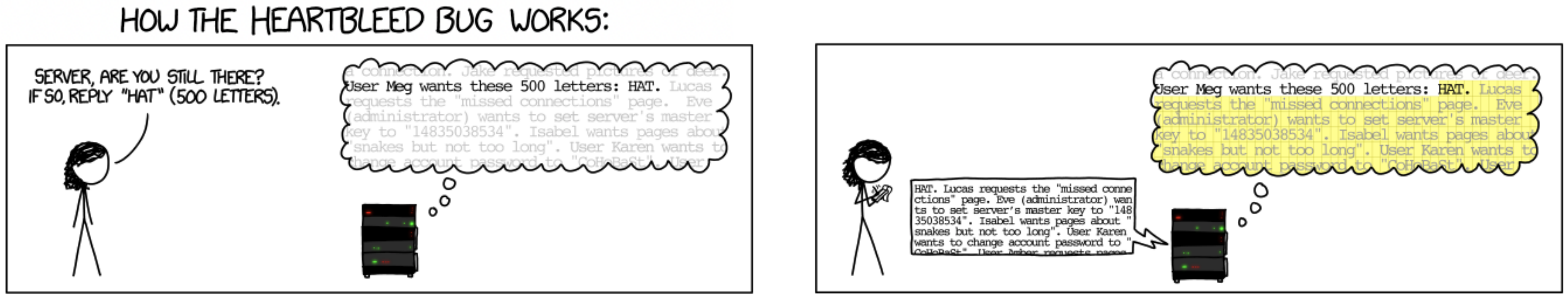
Fuzzing, motivations

- 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

- 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

- We can use the fuzzer and pass 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 if terminated correctly
result.stderr # e.g., '' if no error msgs
```

- 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)
    ```

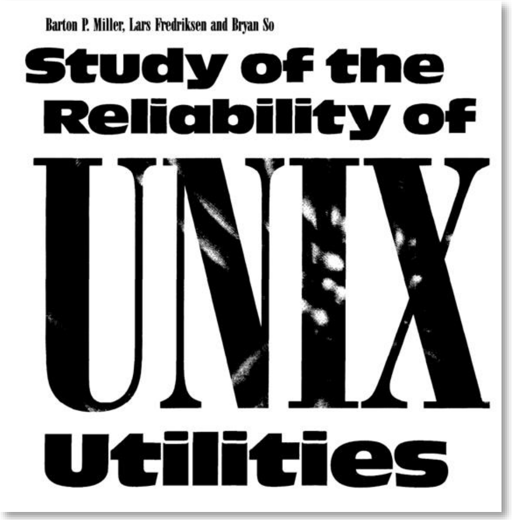
Fuzzing external programs — Warning

- 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

- [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

POLITECNICO
MILANO 1863

- 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

EXTRA MATERIAL

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

EXTRA MATERIAL

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

EXTRA MATERIAL

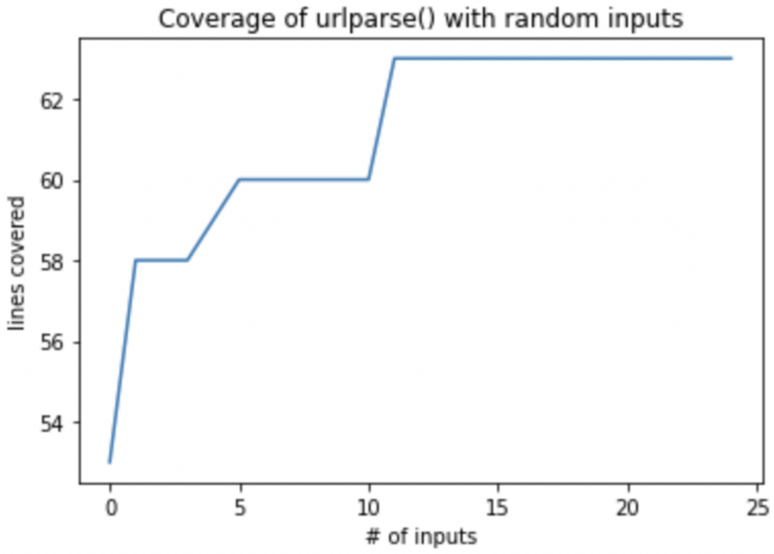
Fuzzing: guiding by coverage

POLITECNICO
MILANO 1863

- 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)
```

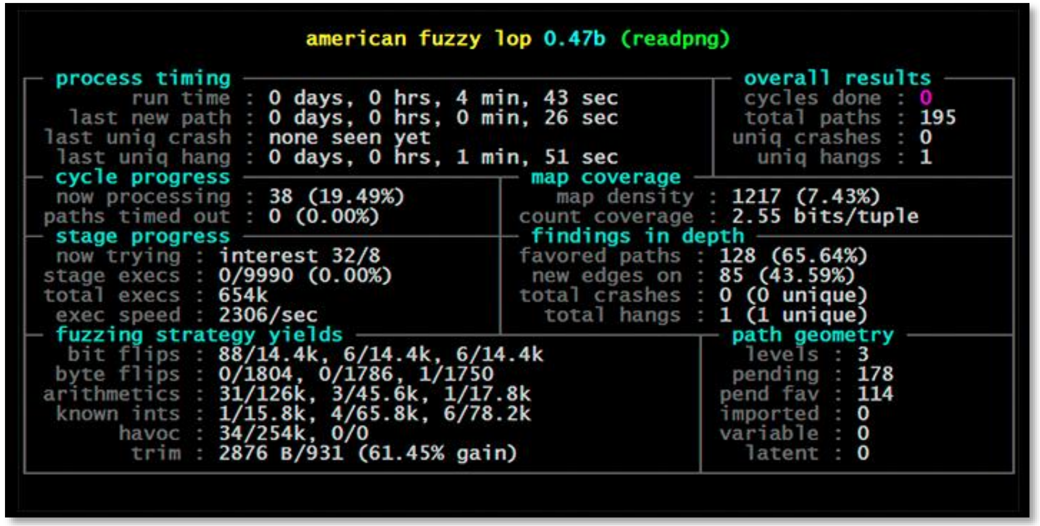
- By inspecting `summary` we can see each and every input is valid and has different coverage!
- Strengths
  - Practical also with large programs – it explores one path after the other until you have budget
  - All you need is a way to capture the coverage



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/

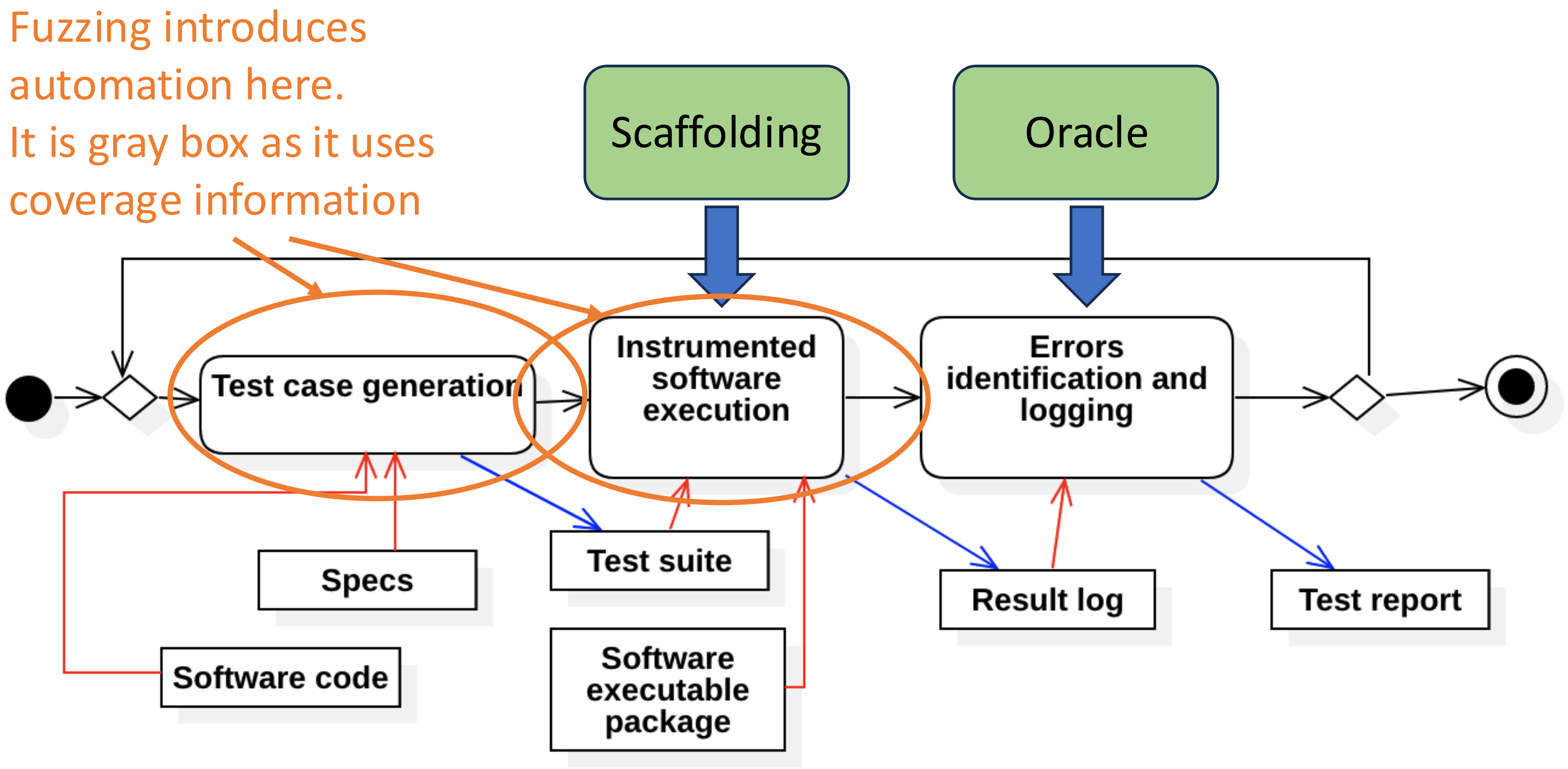
- First to demonstrate that failures and vulnerabilities can be detected automatically in large-scale security-critical applications
- Implements mutational fuzzing + guiding by coverage



AFL User Interface

Positioning fuzzing in the testing workflow

POLITECNICO
MILANO 1863



Verification & Validation

Search-Based Software Testing

Search-Based Software Testing (SBST)

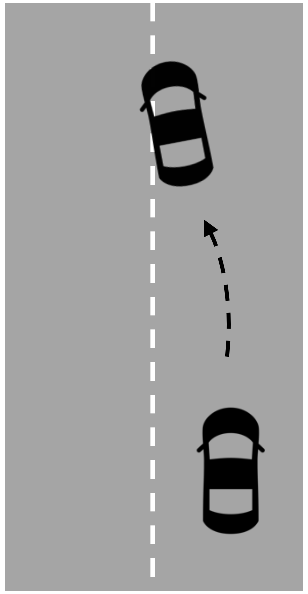
- Complements test case generation techniques seen so far
  - Works at **component** or **system** level
  - Guides the generation toward a **specific testing objective**
  - Compared to fuzzing, typically incorporates **domain-specific knowledge** to generate more **meaningful** test cases
  - Can deal with **functional** and **non-functional** aspects (e.g., reliability, safety)

SBST: the very idea

- Generate **test cases** that achieve some test objective
  - Examples of common test objectives
    - Observing wrong/undesired outputs
    - Breaking given requirements
    - Reaching specific source code locations
    - Executing some given paths
- Essence of SBT
  - Recast testing as an optimization problem
    - **Search space** + **fitness** to guide the exploration
  - Generate better and better tests to achieve the objective

SBST: motivating (toy) example

- Autonomous driving
  - Assume you want to test the subsystem that controls the **steering angle** of an autonomous vehicle
  - **Safety requirement**: *“the vehicle shall always maintain a given safety distance from the centre of the road”*



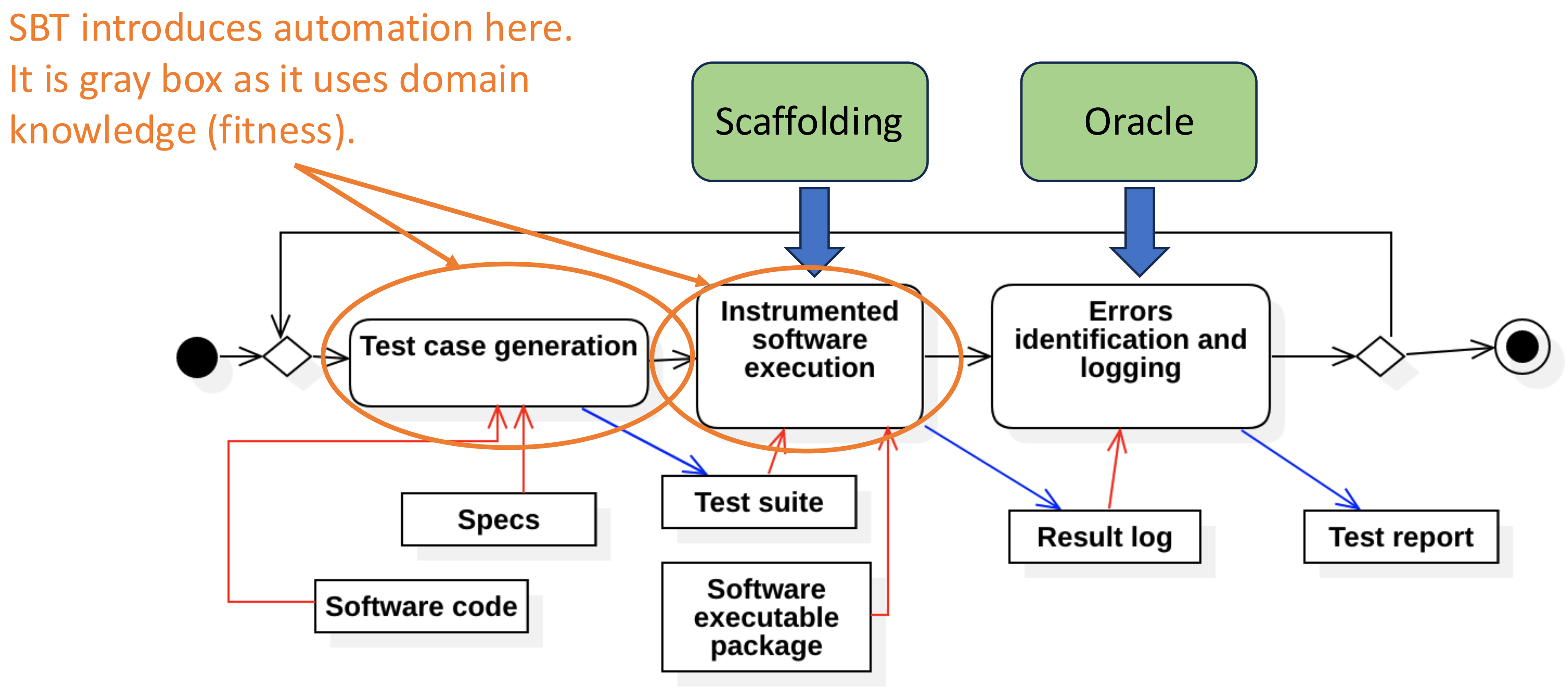
Safety requirement unsatisfied

Safety requirement satisfied

SBST: steps

1. Identify the objective
2. Define how to measure the distance from the objective
    - The distance measures the *fitness* by executing a test case
    - A test case is identified by the inputs provided to the program
3. Instrument the code to compute the fitness
4. Select one or more test cases (e.g., random inputs)
5. Execute the test cases and compute the fitness
6. If fitness is not sufficient, go to step 4
7. else, stop (e.g., found a test case that achieves the objective)

Positioning SBST in the testing workflow



SBT: Summary

- Strengths
  - Compared to concolic guides the generation toward a **specific testing objective** (e.g., wrong outputs, coverage of given branches)
  - Compared to fuzzing typically generate more **meaningful** test cases
  - Eventually reaches the objective with enough budget
- Weaknesses
  - Requires domain-knowledge to define the notion of fitness (nontrivial)
  - Heavily relies on the quality of the heuristics to explore the neighborhood
  - Computationally expensive and time-consuming

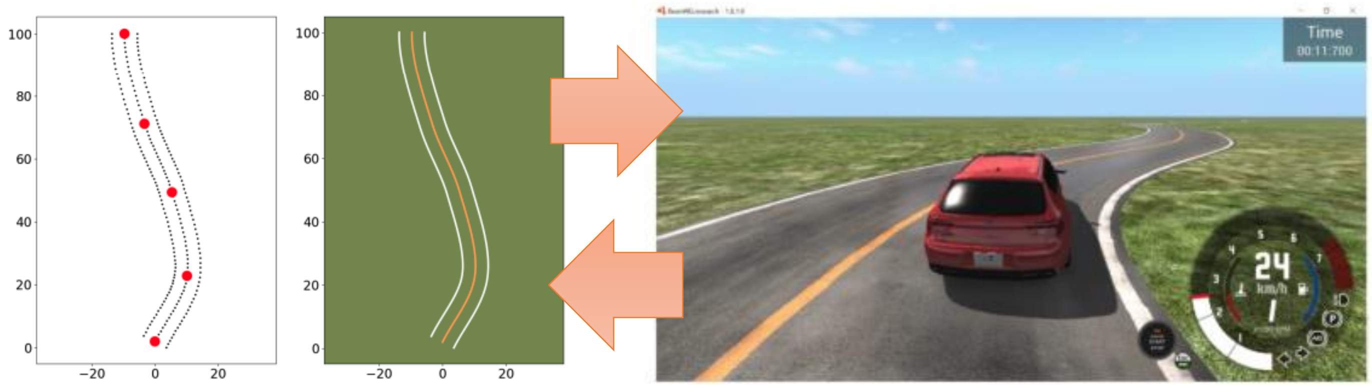
SBST: tools

- EvoSuite
    - Tool that automatically generates test cases for Java code
    - Automatically instruments the sources
    - Generates test suites towards satisfying a given objective (i.e., coverage criterion)
    - https://www.evosuite.org/

EVOSUITE
Automatic Test Suite Generation for Java

SBST: latest trends

- Automated testing of autonomous systems
  - **Example**: search for road shapes where the car cannot keep the lane (testing objective = invalidate safety requirements)



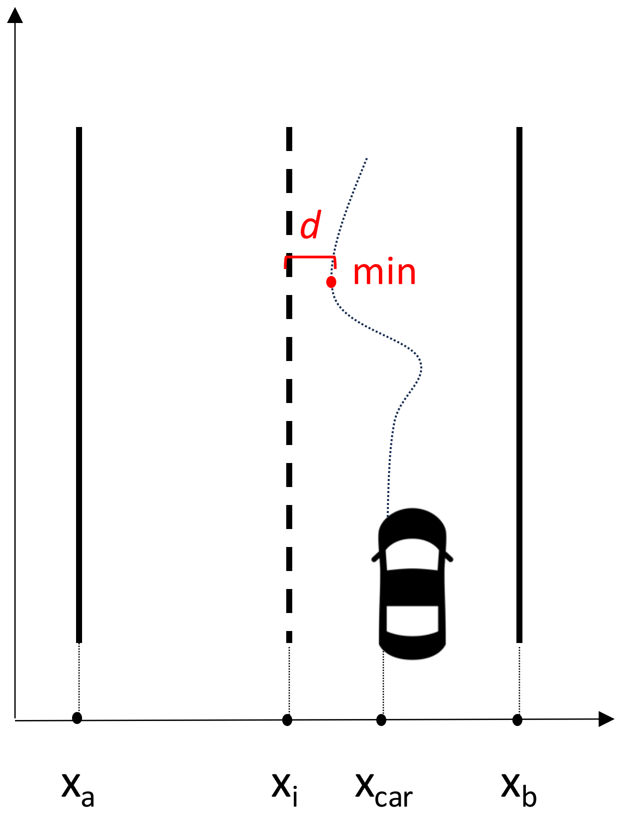
Test case generation
road shape

Scenario simulation
fitness calculation given the generated road

https://dl.acm.org/doi/10.1145/3368089.3409730

EXTRA MATERIAL

SBST Toy example



- Assume we have a simulator for the autonomous vehicle

```
simulate_car(xa,xb,xcar)
```

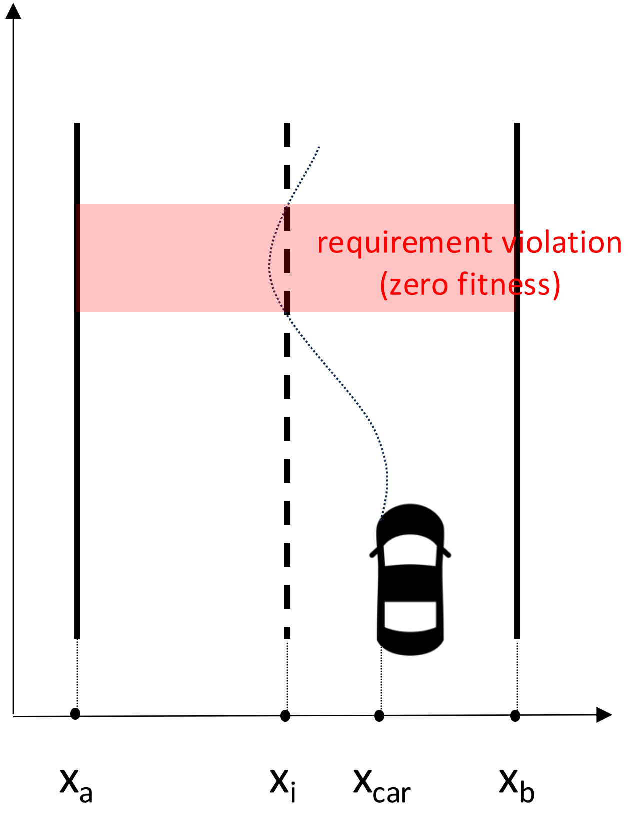
  - **Input**: road description (`xa`, `xb`), initial position of the car (`xcar`)
  - **Output**: trajectory of the car (sequence of values for `xcar`)

- Objective: identify test cases where the trajectory crosses the center of the road x_i
- How to measure the distance *d* from the objective?

```python
def get_distance(xa: int, xb: int, xcar: int):
    xi = (xa+xb)/2
    return min(simulate_car(xa,xb,xcar)) - xi
```

EXTRA MATERIAL

SBST Toy example



- Let's create our fitness function
  - Fitness measures the goodness of the test cases
  - We consider all test cases that meet the objective to be equally good

```
def fitness(xa: int, xb: int, xcar: int):
    r = get_distance(xa, xb, xcar)
    if r<=0:
        return 0
    return r
```

- The lower the fitness, the better — we want to identify test cases that yields **zero fitness**

EXTRA MATERIAL

SBST: key points

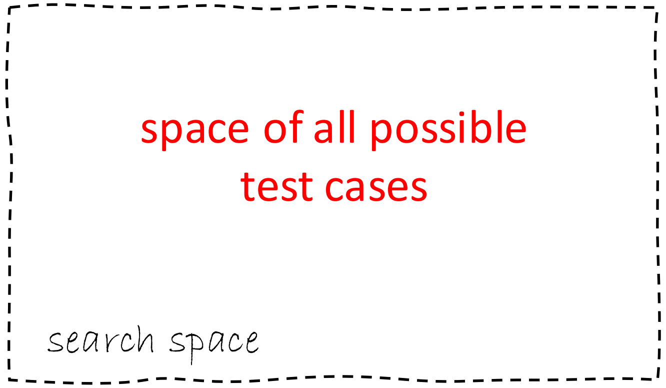
- So far
  - Definition of the objective
  - Definition of the fitness function
  - We instrumented the code
- Next
  - Definition of the selection method for **test case** candidates
    - how do we search in the **space** of test cases?

EXTRA MATERIAL

Test Generation as a Search Problem

POLITECNICO MILANO 1863

- Search space
  - Defines **what we are looking for** (test cases)
  - Depends on the testing problem
    - Single integer values, tuples of values, complex objects, XML documents, …



- Example: system under test `simulate_car`
  - 3 input parameters
  - Search space (representation for test cases) = input tuples (xa, xb, xcar)

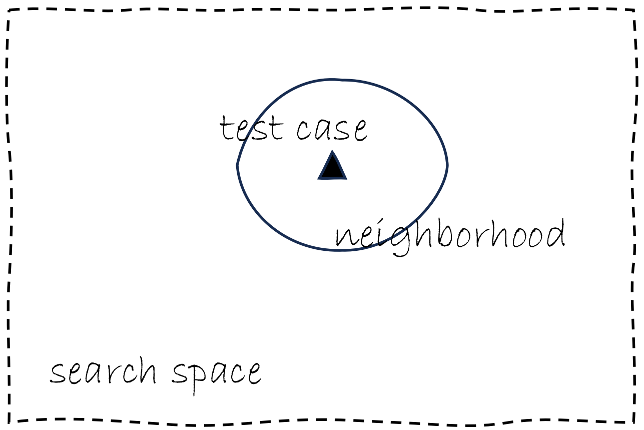
```
simulate_car(3,10,7)
simulate_car(5,15,8)
simulate_car(10,22,19)
```

EXTRA MATERIAL

Test Generation as a Search Problem

POLITECNICO MILANO 1863

- Search space: neighborhood
  - Each point in the search space has its **neighbors**
  - Neighborhood includes inputs that are related to a given one (e.g., “close” according to some distance function)
- Foo example
  - Test case = input tuple (xa, xb, xcar)
  - Each tuple has 26 neighbors (3*3*3 - 1)
    - xa-1, xb-1, xcar-1
    - xa-1, xb-1, xcar
    - xa-1, xb-1, xcar+1
    - xa-1, xb, xcar-1
    - xa-1, xb+1, xcar-1
    - xa, xb-1, xcar-1
    - xa+1, xb-1, xcar-1
    - …

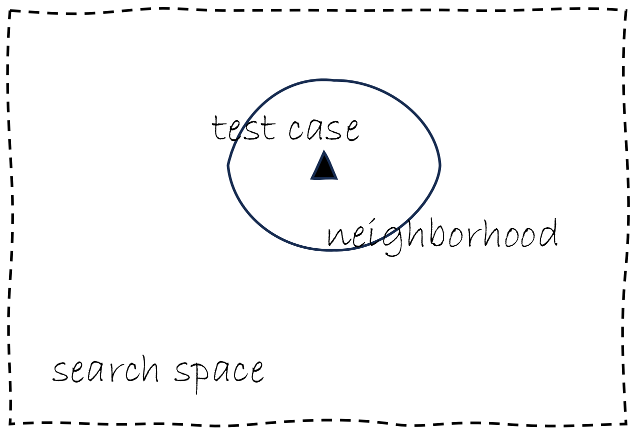


EXTRA MATERIAL

Test Generation as a Search Problem

POLITECNICO MILANO 1863

- Given **search space** + **neighborhood relation** we can define
  - Fitness function → defines the "goodness" of a given test case (candidate solution)
  - Algorithm → explores the neighborhood using the heuristic to steer the search



EXTRA MATERIAL

Test Generation as a Search Problem

- Fitness function
  - "goodness" of an individual = **fitness**
    - Maps a test case to a **numeric value** (the better the candidate the better the value)
  - Depends on the **objective** of testing!

- Example: function `get_distance`

```python
def fitness(xa: int, xb: int, xcar: int):
    r = get_distance(xa, xb, xcar)
    if r<=0:
        return 0
    return r
```

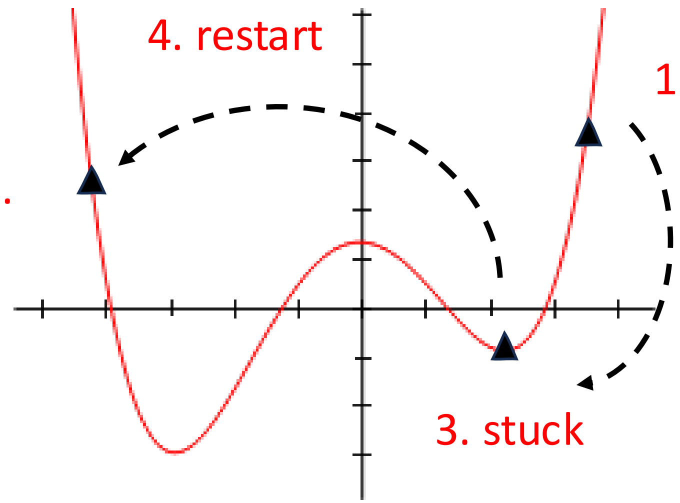
EXTRA MATERIAL

Test Generation: Hillclimbing

- Search algorithm — Hillclimbing (simple meta-heuristic algorithm)
  1. Take a random starting point
  2. Determine fitness value of all neighbors
  3. Move to neighbor with the best fitness value
  4. If solution is not found, continue with step 2

Test Generation: Hillclimbing

- Escaping local optima
  - The easiest way is to give up and restart from a new random point



EXTRA MATERIAL

Test Generation: Evolutionary Search

POLITECNICO MILANO 1863

- Hillclimbing works well if
  - Reasonably **small** neighborhood and search space
- Assume we have a program that receives UTF-16 strings (max len 10) as input:
  - Each char is encoded with 16 bits = 65536 possible encodings
  - What is the size of the search space?
  - Hillclimbing would take **unreasonably long time**! Why?
- Global search
  - Hillclimbing searches locally only!
  - We should make larger steps to extend the search "more globally"

EXTRA MATERIAL

Test Generation: Evolutionary Search

POLITECNICO MILANO 1863

- Global search
  - We can use (again) the notion of **mutation**
  - It can be used to carry out larger steps
    - Shall not completely change an individual
    - Shall keep most of its “traits”
  - **Examples**
    - Mutation for **strings**: flip random char, add/remove a random char, …
    - Mutation for **integers**: sum a small random number, flip some random bits in the binary encoding, …

References

- Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler: "The Fuzzing Book". Retrieved 2023-11. https://www.fuzzingbook.org/
- Fraser, Gordon, and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT symposium and the 13th FSE. 2011. https://dl.acm.org/doi/abs/10.1145/2025113.2025179
- Evo Suite tool: https://www.evosuite.org/
- Vincenzo Riccio and Paolo Tonella. 2020. Model-based exploration of the frontier of behaviours for deep learning system testing. In Proceedings of the 28th ACM ESEC/FSE 2020. ACM, New York, NY, USA, 876–888. https://doi.org/10.1145/3368089.3409730