Lecture 2 – Random Testing and Fuzzing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

2026 Fall

Recall

- Modeling by Hand
  - Equivalence Partitioning (EP) / Boundary Value Analysis (BVA)
  - Category Partition Method (CPM)
- Combinatorial Testing (CT)
  - Covering Arrays – CA / MCA / CMCA
  - Size of Covering Arrays – $v^t \leq N \leq O(t\, v^t \log kv)$
  - Construction – IPOG, and coverage measurement
- Inferring the Input Model
  - from code / documentation / examples + an oracle / a corpus

From a Model to a Sample

Last week: a **model**, covered **deterministically** – every t-way interaction once. Today: drop the guarantee and **sample**.

- **No bias** – the sampler has no mental model of the program.
- **Numbers** – how many tests until a failure, and how sure are we?
- **Speed** – thousands of executions per second, no analysis.

Contents

Contents

Random Testing

- S – the set of all inputs of the **SUT** (software under test)
- $F \subseteq S$ – the inputs that **fail**

failure rate $p = \dfrac{|F|}{|S|}$ – the chance that an input drawn **uniformly** from S fails

```
int abs(int x) {
  if (x < 0) return x;    // should be -x
  else       return x;
}
```

- Every negative `int` fails: $p = 2^{31}/2^{32} = 0.5$.
- **Random testing**: draw inputs from S at random, check each against an **oracle** – here `assertEqual(abs(x), x < 0 ? -x : x)`. Where oracles come from is a later topic; today we assume one is given.

Why Random? And How Random?

PLRG

Why random?

- Developers test the inputs they **thought of**. Random testing has no such bias.
- It is surprisingly **competitive**: about as good as partition testing in simulations,[1] and if a "smart" technique is much **slower** per test, random testing wins in the same time budget.[2]

How random?

- A **pseudo-random** generator with a **seed**: the same seed replays the same run, so a failure is **reproducible**.
- A **biased** generator silently changes p – PHP's `mt_rand` was biased for two years.[3]

---

[1] **[TSE'84]** J. W. Duran, S. C. Ntafos. "*An Evaluation of Random Testing.*"

[2] **[TSE'15]** M. Böhme, S. Paul. "*A Probabilistic Analysis of the Efficiency of Automated Software Testing.*"

[3] **[PHP Bug #75170]** "*mt_rand() bias on 64-bit machines.*"

Two Questions, One Distribution

PLRG

1. Given p, **how many** inputs until the **first failure**?
2. Given n inputs, what is the **probability** of **at least one** failure?

Each draw is an independent Bernoulli trial that "succeeds" – fails the SUT – with probability p. The number of draws X up to the first success is **geometric**:

$$\Pr(X = k) = \underbrace{(1-p)^{k-1}}_{k-1 \text{ passes}} \underbrace{p}_{\text{then a failure}}$$

$p = 0.01$:

k	1	2	3	10	69	100
$\Pr(X = k)$	.0100	.0099	.0098	.0091	.0050	.0037

The single most likely value is $k = 1$. So how many draws should we **expect**?

How Many Inputs Until the First Failure?

PLRG

$$E(X) = \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k\,(1-p)^{k-1} p$$

$$= p\left(\sum_{k=1}^{\infty}(1-p)^{k-1} + \sum_{k=2}^{\infty}(1-p)^{k-1} + \sum_{k=3}^{\infty}(1-p)^{k-1} + \cdots\right)$$

$$= p\left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \cdots\right)$$

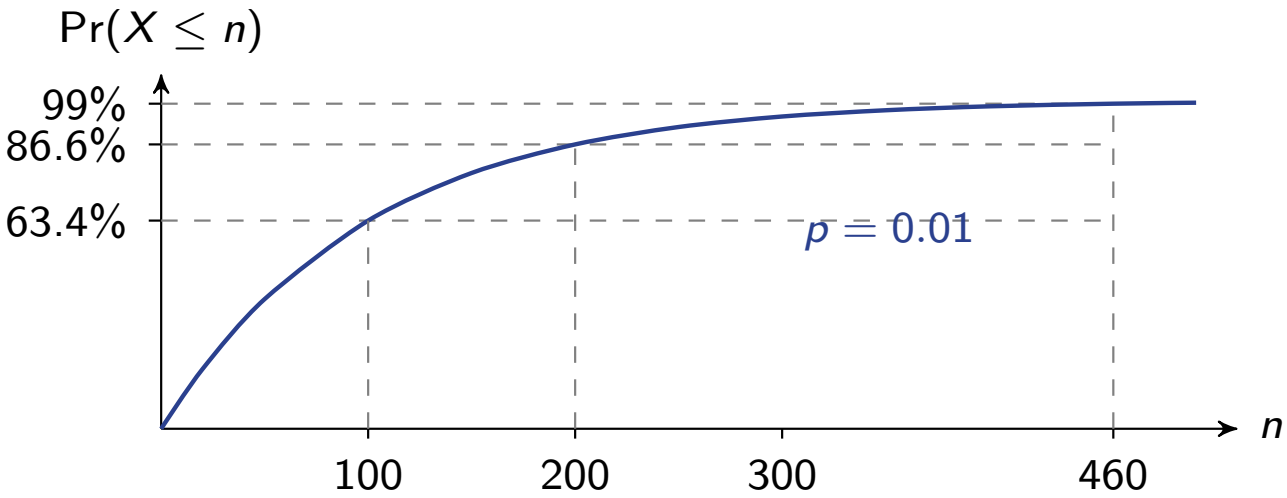
$$= 1 + (1-p) + (1-p)^2 + \cdots = \boxed{\frac{1}{p}}$$

	formula	$p = 0.01$
mean	$1/p$	100
median	$\lceil -1/\log_2(1-p) \rceil$	69
standard deviation	$\sqrt{1-p}\,/\,p$	≈ 99.5

The standard deviation is as large as the mean: one run may need 10 tests, the next 300. “100 on average” is a weak promise.

How Likely Is at Least One Failure in n Inputs?

$$\Pr(X \le n) = \sum_{k=1}^{n} (1-p)^{k-1} p = p \frac{1-(1-p)^n}{1-(1-p)} = \boxed{1-(1-p)^n}$$



- Running the **mean** number of tests, $n = 1/p = 100$, finds the failure only 63% of the time (in general $1-(1-p)^{1/p} \approx 1-1/e$).
- To find it with probability c, solve $1-(1-p)^n \ge c$ for n:

$$n \ge \frac{\ln(1-c)}{\ln(1-p)} \approx \frac{\ln \frac{1}{1-c}}{p} \qquad c = 99\%: \quad n \approx \frac{\ln 100}{p} = \frac{4.6}{p} = 460$$

  – 99% confidence costs **4.6 times** the mean.

But p Is Unknown – Reading n Clean Runs

In practice nobody gives us p. What we see is: n **random inputs ran, none failed.** Does that mean $p = 0$?

- No – a **small** p also gives n clean runs, by luck. Turn it around: **if** the rate were at least ε, how likely are n clean runs?

$$\Pr[\, n \text{ clean runs} \mid p \geq \varepsilon \,] \;=\; (1-p)^n \;\leq\; (1-\varepsilon)^n \;\leq\; e^{-\varepsilon n}$$

- Concretely, $\varepsilon = 0.5\%$ and $n = 600$:

$$e^{-0.005 \cdot 600} = e^{-3} \approx 0.05$$

  If the rate were 0.5% or worse, 600 clean runs would happen only **5% of the time**. We saw them – so, **95% confident**, $p < 0.5\%$.
- Two numbers, two roles: the **threshold** (0.5%) is what we rule out; the **confidence** (95%) is how sure we are.

Residual Risk

In general, pick confidence $1-\delta$ you want, set $e^{-\varepsilon n} = \delta$, and solve for ε:

after n clean runs, with confidence $1-\delta$: $\quad p < \dfrac{\ln(1/\delta)}{n}$

clean runs n	1,000	10,000	1,000,000
95% confidence ($\ln 20 = 3.0$, so $p < 3.0/n$)	0.003	0.0003	3×10^{-6}
99% confidence ($\ln 100 = 4.6$, so $p < 4.6/n$)	0.0046	0.00046	4.6×10^{-6}

- The "rule of three": after n clean runs, $p < 3/n$ with 95% confidence. Ten times more tests buy one more decimal place – and **never** $p = 0$.
- This bound is the **residual risk** of a testing campaign: how bad the failure rate could still be. It assumes **independent** draws – for a fuzzer, whose draws are not, estimating it is recent research.[4]

[4] **[FSE'21]** M. Böhme, D. Liyanage, V. Wüstholz. "*Estimating Residual Risk in Greybox Fuzzing.*"

Needles in a Haystack

```c
/* C, 32-bit int */
void foo(int x) {
  if (x == 0) {
    /* faulty code here */
  }
}
```

```python
# Python, x in [0, 2**32)
def foo(x):
  # e.g., x = 2840
  if x * 7919 % 5711 == 42:
    # faulty code here
```

	p	$E(X) = 1/p$
`x == 0`	$2^{-32} \approx 2.3 \times 10^{-10}$	4.3 billion
`x * 7919 % 5711 == 42`	$1/5{,}711$	5,711

- Uniform random gets the second in a blink and **never** gets the first. The needle's size is p, not how hard the code looks.
- Hence **biased** random: **special values** (0, -1, `INT_MAX`, `null`, `NaN`, `""`), **constants** from the code, values that **failed before**. Fuzzers call this list "interesting values" and a **dictionary**.

Contents

Random Testing an API

So far an input was a **value**. For a library, a test input is a **sequence of method calls** that builds up objects and then calls the method under test:

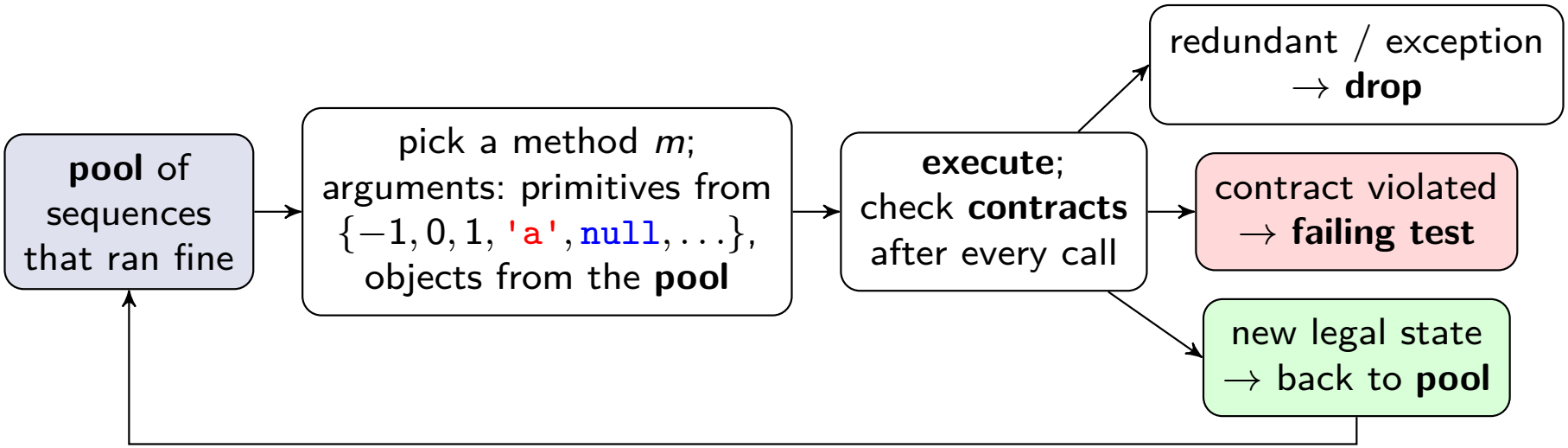
```
LinkedList l = new LinkedList();
l.addFirst(new Object());
TreeSet    t = new TreeSet(l);          // <- the method under test
```

- Plain random generation picks methods and arguments **blindly**. Two things go wrong:
  - **Illegal prefixes.** After a call that already misbehaved, every extension is wasted:

```
double a = Math.sqrt(-1);    // NaN -- illegal argument
double b = Math.log(a);      // pointless: a is already garbage
```

  - **Redundant prefixes.** `new LinkedList()` a thousand times gives the same empty list a thousand times.
- Idea: **run** each sequence as soon as it is built, and let the **result** decide whether it is worth extending.

Randoop – Feedback-Directed Random Testing



- **Build** one sequence at a time: a random method, arguments from a small pool of primitives and from objects **earlier sequences** produced.
- **Execute** it immediately. Three outcomes:
  - a **contract** such as `o.equals(o)` failed → a failing test;
  - an exception, or an object seen before → drop it;
  - otherwise → a **building block** for longer sequences.
- That third outcome is the **feedback**:[5] only sequences that ran **legally** and reached a **new** state are extended.

[5] **[ICSE'07]** C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. "*Feedback-Directed Random Test Generation.*"

What Randoop Found

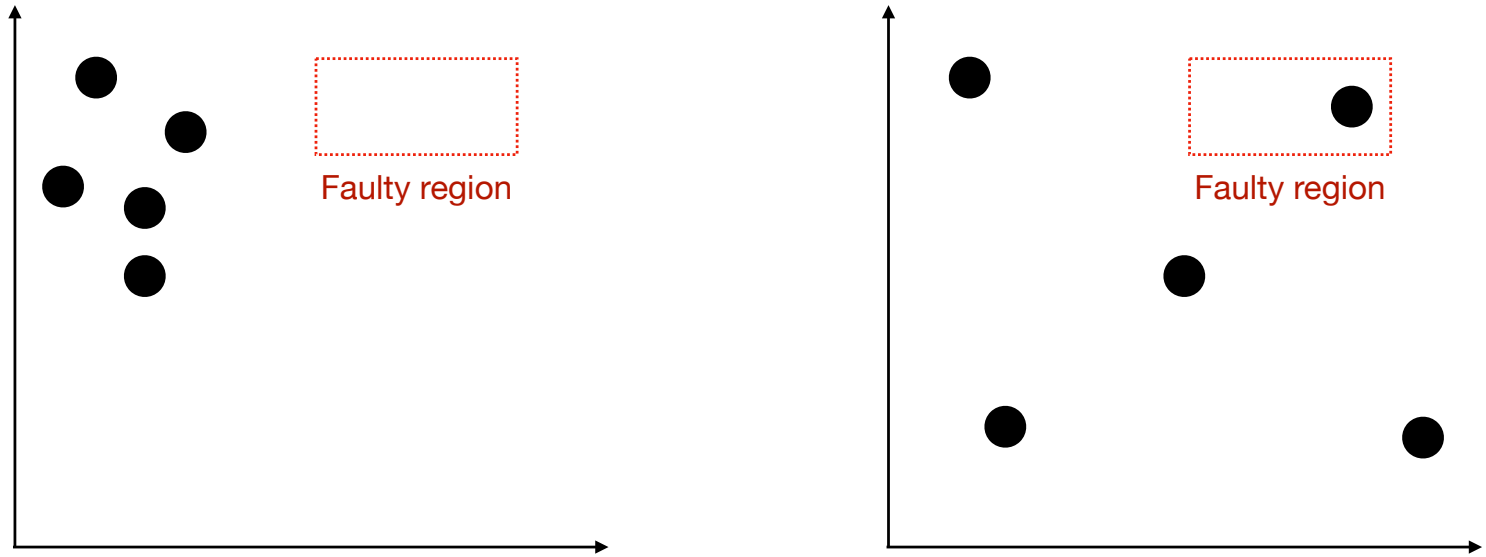
A test Randoop generated for `java.util` (JDK 1.5), with our comments:

```
LinkedList l1 = new LinkedList();
Object     o1 = new Object();      // a plain object: cannot be compared
l1.addFirst(o1);
TreeSet    t1 = new TreeSet(l1);   // sorted set: must reject o1 (bug 1)
Set        s1 = Collections.unmodifiableSet(t1);      // read-only view
assertTrue(s1.equals(s1));         // false: not equal to itself (bug 2)
```

- The only rule Randoop checked: **every object equals itself**.
- 14 libraries, 780 KLOC, many new bugs – still the classic baseline for unit-test generators.

Contents

Failures Cluster



- A fault guarded by `x >= 0 && x < 100` fails on a **contiguous** region – a **faulty region**. Empirically, failing inputs form **blocks**, **strips**, or scattered **points**.[6]
- Two tests **close together** likely give the **same** verdict. So: keep sampling at random, but **spread the samples out**.

[6] **[JSS'10]** T. Y. Chen, F.-C. Kuo, R. G. Merkel, T. H. Tse. "*Adaptive Random Testing: The ART of Test Case Diversity.*"

Fixed-Size-Candidate-Set ART (FSCS-ART)

The most common form of ART.[7] T is the set of tests **executed so far**.

1. $T \leftarrow \{\text{one random input}\}$
2. **Repeat** until the stopping criterion (budget, or a failure):
   1. Draw Z **candidates** at random – $Z = 10$ in the original.
   2. For each candidate c, find the **nearest** executed test and take that distance:

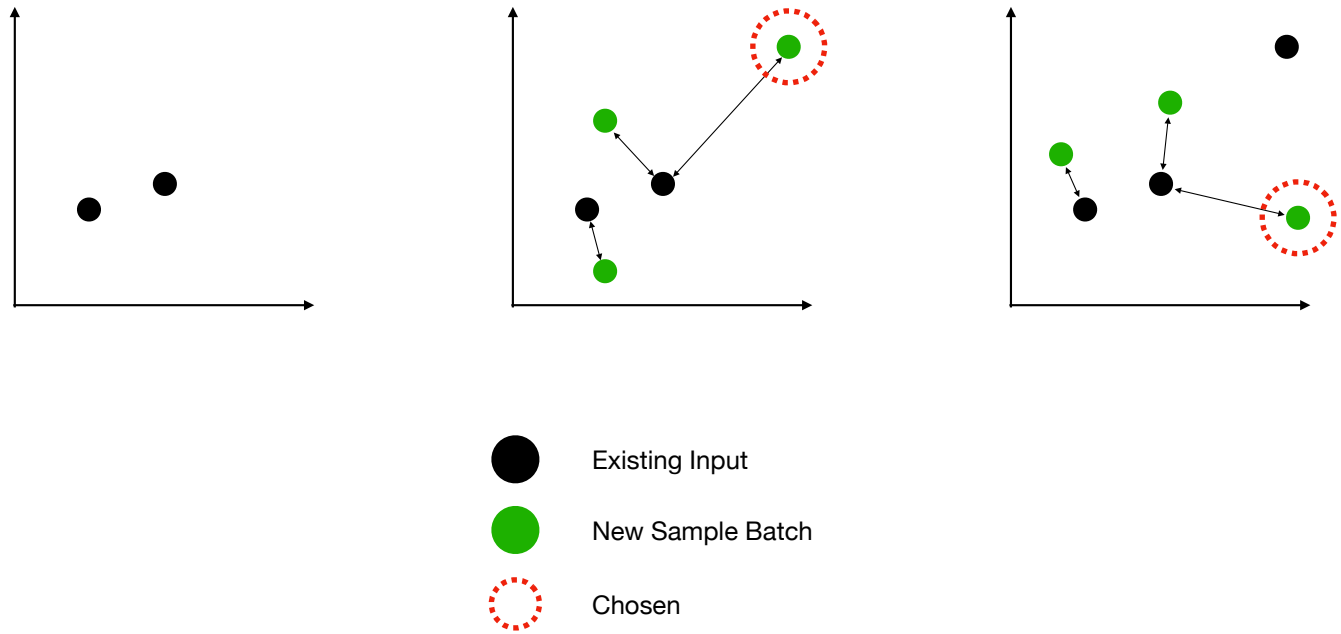
$$fitness(c) = \min_{t \in T} d(c, t)$$

   3. **Execute** the candidate with the **largest** fitness – the one farthest from everything tried so far – and add it to T. The other $Z - 1$ candidates are thrown away.

Still random – every candidate is a random draw – but a **distance** decides which draw is executed.

[7] **[ASIAN'04]** T. Y. Chen, H. Leung, I. K. Mak. "*Adaptive Random Testing.*"

FSCS-ART – One Step



- Black: tests already executed. Green: $Z = 3$ fresh random candidates.
- Each arrow is a candidate's distance to its **nearest** black point. The candidate with the **longest** arrow – the one in the red dashed circle – is executed; the other two are discarded. Right: the next round, with the chosen point now black.
- Variants replace "nearest" by the **average** or the **centroid** of T; the shape is the same – random candidates, a **distance** decides.

Distance Between Inputs

Numbers: **Euclidean** $d(x, y) = \sqrt{\sum_i (x_i - y_i)^2}$. Strings: the **Levenshtein (edit) distance** – the fewest single-character **insertions**, **deletions**, **substitutions**:

$$\text{“kitten”} \xrightarrow[k \to s]{\text{substitute}} \text{“sitten”} \xrightarrow[e \to i]{\text{substitute}} \text{“sittin”} \xrightarrow[g]{\text{insert}} \text{“sitting”} \qquad d = 3$$

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

$$D[i][j] = \min\{\, D[i-1][j] + 1,\; D[i][j-1] + 1,\; D[i-1][j-1] + [a_i \neq b_j] \,\}$$

dynamic programming, $O(mn)$
– and essentially optimal.[8]

Objects: ARTOO[9] mixes **type** and **field** distances, recursively.

[8] **[STOC'15]** A. Backurs, P. Indyk. “*Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false).*”

[9] **[ICSE'08]** I. Ciupa, A. Leitner, M. Oriol, B. Meyer. “*ARTOO: Adaptive Random Testing for Object-Oriented Software.*”

Is ART Worth It?

cost	$Z(0+1+\cdots+(k-1)) = \frac{k(k-1)}{2}Z \implies O(k^2Z)$ distance computations
gain	F-measure (tests to first failure): $1/p$ for RT $\implies$ at best $1/2p$ for **any** strategy

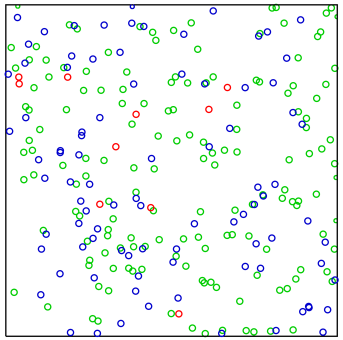
- Chen & Merkel[10] – **no** strategy beats random testing's F-measure by more than **50%**. ART can at best **halve** the number of tests.
- Arcuri & Briand[11] – real failure rates are tiny, so k is huge, so $O(k^2Z)$ eats the gain: by **wall-clock**, plain random testing often wins.
- The debate goes on.[12] The durable lesson is the **idea**: diversity as feedback.

[10] **[TOSEM'08]** T. Y. Chen, R. Merkel. "*An Upper Bound on Software Testing Effectiveness.*"

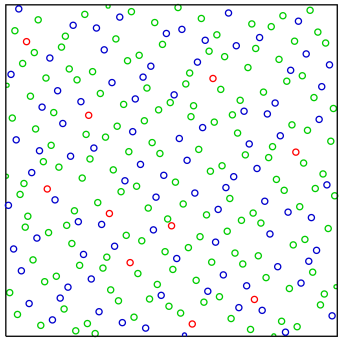
[11] **[ISSTA'11]** A. Arcuri, L. Briand. "*Adaptive Random Testing: An Illusion of Effectiveness?*"

[12] **[TSE'19]** R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, X. Xia. "*A Survey on Adaptive Random Testing.*"

Quasi-Random Testing – Diversity for Free



pseudo-random



quasi-random

Random points leave **gaps** and **pile up**. A **quasi-random** sequence is computed to spread **evenly** – ART's goal, with no distance computations.[13]

Halton sequence: keep splitting $[0, 1]$ into 2 (x) and 3 (y); pair the cut points in order:

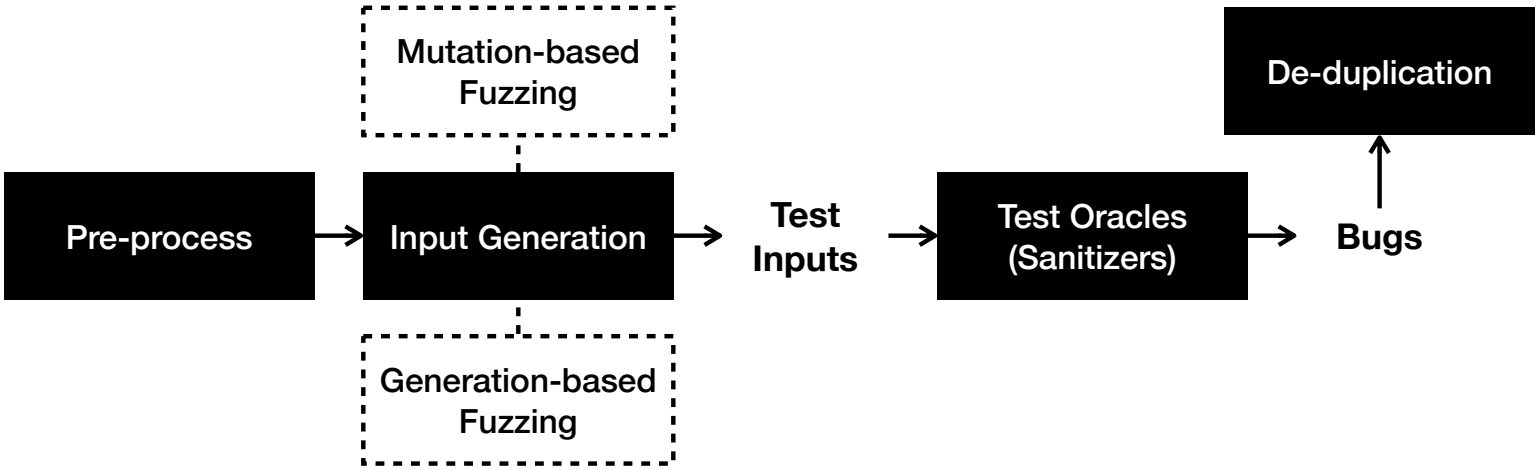
$$\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \cdots \qquad \frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{4}{9}, \frac{7}{9}, \frac{2}{9}, \frac{5}{9}, \cdots$$

$$(\frac{1}{2}, \frac{1}{3}), (\frac{1}{4}, \frac{2}{3}), (\frac{3}{4}, \frac{1}{9}), (\frac{1}{8}, \frac{4}{9}), (\frac{5}{8}, \frac{7}{9}), (\frac{3}{8}, \frac{2}{9}), \cdots$$

[13] **[IEEE Trans. Reliability'07]** T. Y. Chen, R. Merkel. "*Quasi-Random Testing.*"

Contents

Fuzzing in One Picture



Fuzzing = make an input, run, watch for a crash, repeat.[14]

- **Where do inputs come from? – generated**, or **mutated**.
- **Which inputs to keep?** – those that did something **new**.
- **What is a bug?** – a crash; **sanitizers** make silent errors crash.
- **Which crashes are the same bug? – de-duplication**.

[14] **[TSE'19]** V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, M. Woo. "*The Art, Science, and Engineering of Fuzzing: A Survey.*"

Generation-Based Fuzzing

Generate inputs from a **model** of what the program expects. The simplest model: fields and their ranges.

```
Date  := Year "-" Month "-" Day
Year  :  int, 1 .. 9999
Month :  int, 1 .. 12
Day   :  int, 1 .. 31
```

Pick each field at random – with the **boundaries** we chose by hand last week now picked automatically:

```
2024-02-30      0001-12-31      9999-01-01      2024-00-15
```

Generation From a Grammar

A grammar for arithmetic expressions:

```
E ::= E + E | E * E | ( E ) | N        N ::= 0 | 1 | ... | 9
```

$$E \rightarrow E * E \rightarrow (\ E\ ) * E \rightarrow (\ E + E\ ) * E \rightarrow \cdots \rightarrow (3 + 4) * 7$$

- At every nonterminal, pick a production **at random**. Two knobs: **weights** on the alternatives, and a **depth limit** so the recursion stops.

- Every output is **syntactically valid**, so it gets past the parser for free. The price: someone has to write the model.

Syntactically Valid Is Not Enough: CodeAlchemist

A grammar gives JavaScript that **parses** – and then dies at runtime:

```
b.length;                       // ReferenceError: b is not defined
var a = 1;  a.push(2);          // TypeError: a.push is not a function
```

CodeAlchemist[15] cuts seed programs into **code bricks**, each with the variables it **uses** and **defines** – and their **types**, inferred by running the seeds:

```
var x = [1, 2];         // defines x : Array
x.push(3);              // uses    x : Array
x.charAt(0);            // uses    x : String
```

- Glue bricks together only when they are **compatible**: brick 1 then brick 2 – yes; brick 1 then brick 3 – never.
- Valid syntax **and** semantics, so it runs deep: 19 bugs in four engines, 11 of them security bugs.

[15] **[NDSS'19]** H. Han, D. Oh, S. K. Cha. "*CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.*"

A Generator With Semantics: Csmith

For a C compiler a grammar is not enough: a random program with **undefined behavior** proves nothing – any output is legal. Csmith[16] builds programs that **cannot** misbehave – every operation is wrapped:

```
static int32_t safe_add_int32(int32_t a, int32_t b) {
  return ((b > 0 && a > INT32_MAX - b) || (b < 0 && a < INT32_MIN - b))
         ? a : a + b;
}
```

- Every generated program has **one** correct output, so two compilers that **disagree** expose a bug.
- Three years of Csmith: **325+** bugs in GCC and LLVM; every compiler tested both crashed and **silently miscompiled**.

[16] **[PLDI'11]** X. Yang, Y. Chen, E. Eide, J. Regehr. "*Finding and Understanding Bugs in C Compilers.*"

Probabilistic Grammar Fuzzing

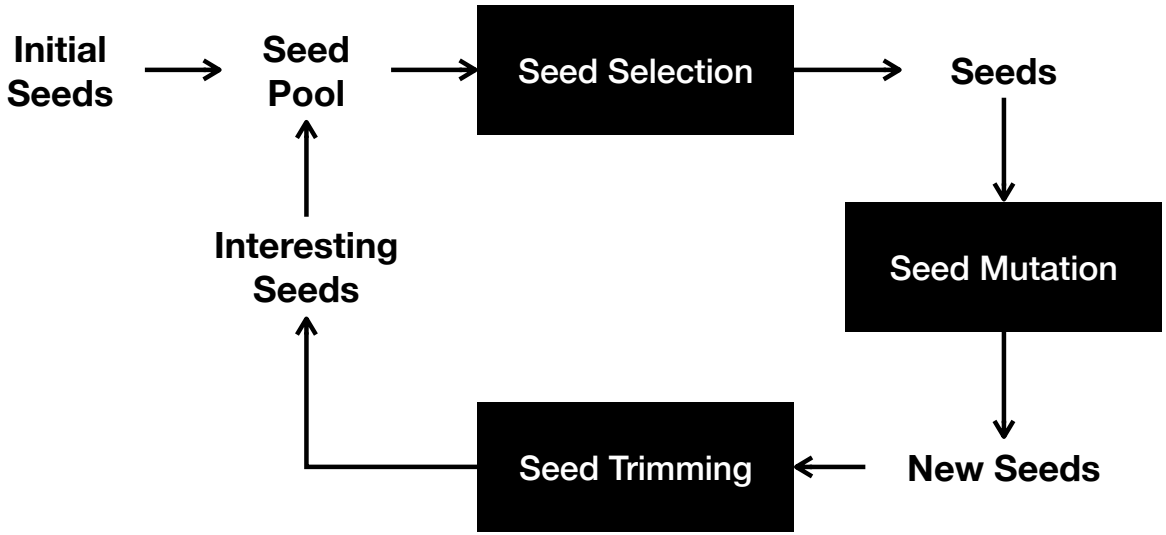
A grammar says what is **possible**; a corpus says what is **common**. Count how often real inputs use each production and attach the counts:

```
E ::= E + E   (45%)  |  E * E   (30%)  |  ( E )   (5%)  |  N   (20%)
```

- Sample with these weights → inputs that look like **real** ones, so they pass the checks real inputs pass.
- **Invert** the weights → inputs made of the **rare** productions – the corners nobody exercised (here: parentheses).
- Skyfire[17] learns such a grammar from thousands of XML/XSL samples and uses it to generate **seeds** for a mutation-based fuzzer – the two halves meet.

[17] **[S&P'17]** J. Wang, B. Chen, L. Wei, Y. Liu. "*Skyfire: Data-Driven Seed Generation for Fuzzing.*"

Mutation-Based Fuzzing



- No model. Start from **seeds** – inputs the program accepts. Pick one, **mutate** it a little, run it, repeat.
- A slightly mutated seed is still **mostly valid**, so it gets **deep** into the program before anything unusual happens.

Mutating Bytes

The seed is just bytes; the fuzzer does not know it is a date:

```
2024-06-15
```

- **Flip** a bit: 2024-06-1u ('5' = 0x35 → 0x75)
- **Add** or **subtract** a small number from a byte: 2024-06-16, 2024-06-55
- **Delete**, **insert**, or **copy** a block of bytes: 2024-06, 2024-06-15-15, 202406-15
- **Paste** a dictionary token: 2024-06--1, 2024-06-2147483647 (Section 1's special values)
- **Splice** with another seed: 2024- + 12-31 from 1999-12-31

Every operator works on byte **positions**. None knows that - separates fields – when a mutant happens to respect the format, that is **luck**. Cheap, and the same code fuzzes a PNG, a PDF, or a date.

Mutating Structure: LangFuzz

Byte mutation of **source code** mostly produces syntax errors. LangFuzz[18] mutates at the level of the **grammar** instead. Two tests from the engine's test suite:

```
// test A                                  // test B
var a = [1, 2, 3];                         var o = {};
a.length = 0;                              o.__proto__ = null;
```

Cut both into **fragments** (statements, expressions, . . .). Replace a fragment of A with one of the **same kind** from B, renaming variables to ones in scope:

```
var a = [1, 2, 3];
a.__proto__ = null;                        // B's statement, o -> a
```

- Always parses; combines features no single test combined. Three months on Mozilla's engine: **105** security bugs.

[18] **[USENIX Security'12]** C. Holler, K. Herzig, A. Zeller. "*Fuzzing with Code Fragments.*"

Mutating Around What Matters: DIE

PLRG

This seed is valuable because its **hot loop** gets the array **JIT-compiled**. Replacing a fragment may throw that away:

```
var a = [1, 2, 3];
for (var i = 0; i < 100; i++) a.push(i);    // replace this line ...
a.length = 0;                               // ... and the JIT is gone
```

DIE[19] keeps two **aspects** of the seed – its **structure** (the loop) and its **types** (a an array, `i` a number) – and mutates only the rest:

```
var a = [7, -1, 2.5];                       // same type, new values
for (var i = 0; i < 250; i++) a.push(i);    // loop kept, bound changed
a.length = 1e9;                             // number -> number
```

- Still a hot loop over an array: the mutant reaches the **JIT**. 2.4× fewer runtime errors, 5.7× more unique crashes than earlier JS fuzzers; 48 bugs, 12 CVEs.

[19] **[S&P'20]** S. Park, W. Xu, I. Yun, D. Jang, T. Kim. "*Fuzzing JavaScript Engines with Aspect-preserving Mutation.*"

Coverage-Guided Fuzzing

Which mutants are worth keeping as new seeds? **Measure coverage**: keep a mutant if it ran code that **no earlier input** ran.[20]

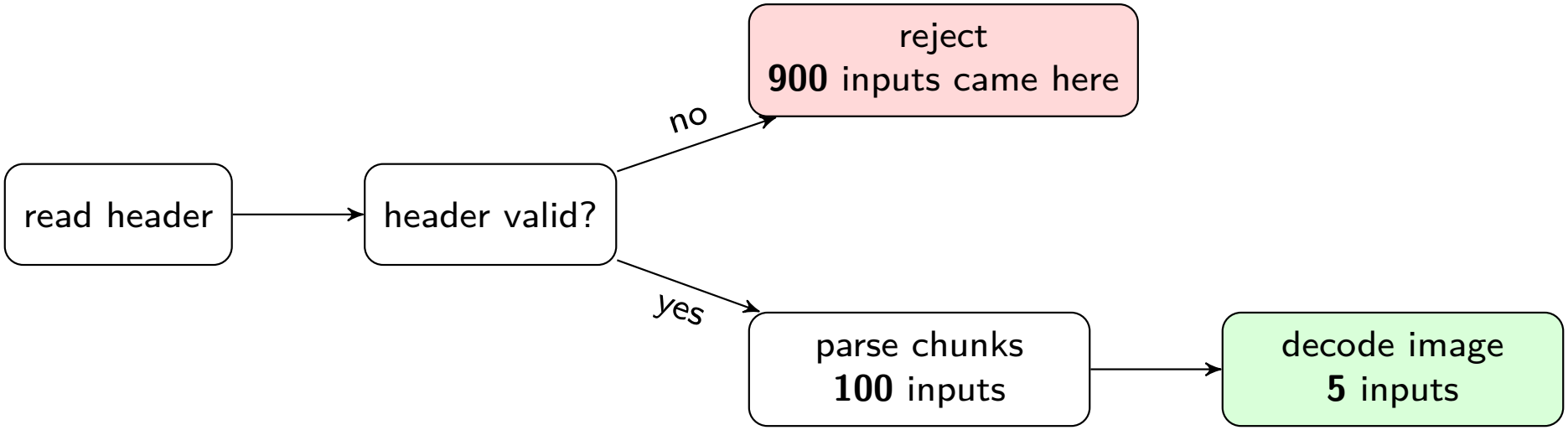
```
if (buf[0] == 'P')
  if (buf[1] == 'N')
    if (buf[2] == 'G')
      crash();
```

- Blind random must guess all **three** bytes at once.
- A mutant that matches `buf[0]` reaches a **new branch**, so it is **kept** – and the next byte is attacked from there. One big needle became three small ones.
- "Coverage" here is a bitmap of **branches taken**, filled by a few instructions the compiler inserts at every branch. What else can count as coverage comes later in the course.

[20] **[2013]** M. Zalewski. "*American Fuzzy Lop.*" **[WOOT'20]** A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse. "*AFL++: Combining Incremental Steps of Fuzzing Research.*"

Beyond Coverage 1 – Rarity

Coverage decides **which** mutants to keep, not **how many** mutations each seed gets. Count where the inputs so far have gone:

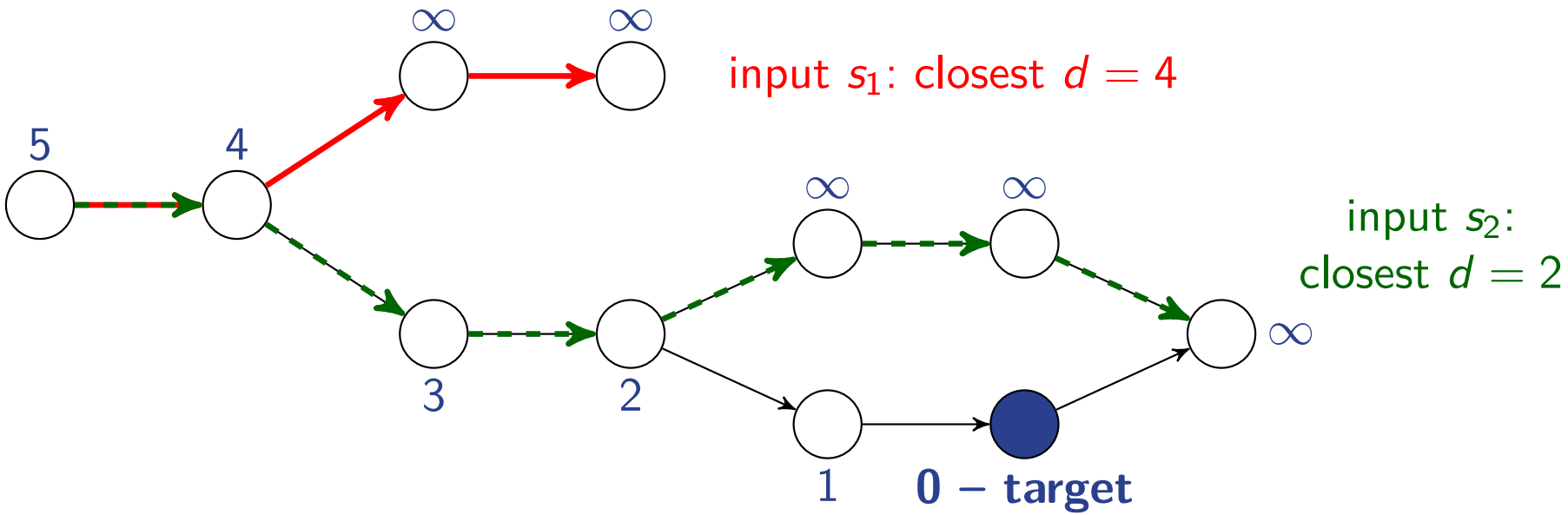


- Most seeds sit on the **common** path (900 of 1,005). Picking seeds uniformly means mutating the **rejected** inputs again and again.
- **Rarity**: a seed on a path that **few** inputs reached gets **more** mutations – its neighborhood is the least explored. AFLFast's power schedule[21] sets energy inversely to the count: here $\times 1$, $\times 9$, $\times 180$.

[21] **[CCS'16]** M. Böhme, V.-T. Pham, A. Roychoudhury. "*Coverage-based Greybox Fuzzing as Markov Chain.*"

Beyond Coverage 2 – Distance (Directed Fuzzing)

Sometimes we know **where** the bug should be – a patched line, a function in a crash report. Mark it as the **target** on the control-flow graph:



- Each node: its **shortest-path distance** d to the target, computed once. An input's distance: the **smallest** d it reached.
- s_2 is closer $\rightarrow$ more mutations. That is the whole change from coverage-guided to **directed** fuzzing.

Directed Fuzzing – Explore First, Then Exploit

AFLGo[22] does not trust the distance from the start: a far input's mutant may be the one that gets close. The weight of distance **grows over time**:

mutations for a seed that is ...	nearest	middle	farthest
at the start	$\times 1$	$\times 1$	$\times 1$
later	$\times 27$	$\times 1$	$\times 1/27$

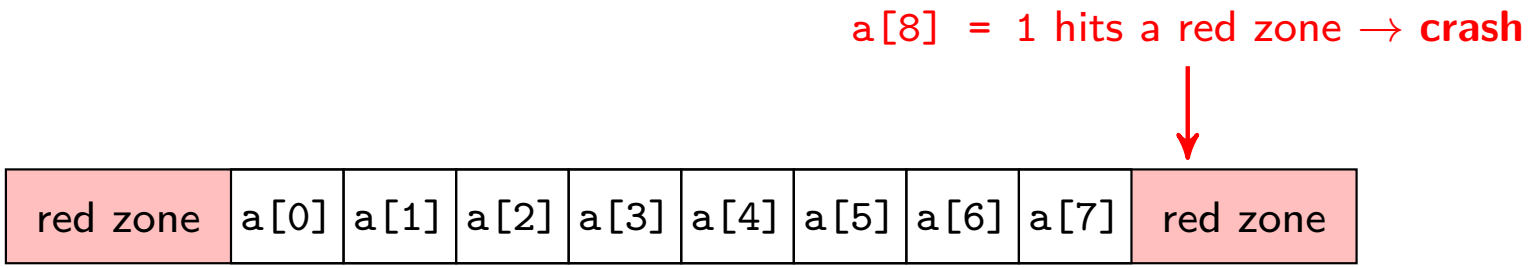
- Early: plain coverage-guided fuzzing (**explore**). Late: almost all mutations go to the inputs nearest the target (**exploit**) – simulated annealing, which we meet again later.
- **Heartbleed**, target = OpenSSL's patched lines: reproduced in about **20 minutes** (mean of 30 runs); directed symbolic execution did not within 24 hours.

[22] **[CCS'17]** M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury. "*Directed Greybox Fuzzing.*"

What Counts as a Bug? Sanitizers

Most memory bugs do **not** crash. Writing one past the end of an array just scribbles on whatever lies next to it – and the program keeps running:

```
int *a = malloc(8 * sizeof(int));
a[8] = 1;          // one past the end -- no crash
```



- AddressSanitizer:[23] put **red zones** – bytes that must never be touched – around every array, and check every read and write. Touch a red zone → crash **immediately**. Now the fuzzer can see the bug.

[23] **[USENIX ATC'12]** K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov. "*AddressSanitizer: A Fast Address Sanity Checker.*"

Triage: Which Crashes Are the Same Bug?

A day of fuzzing leaves thousands of crashing inputs and a handful of bugs. Group them by the **top** n **stack frames** ($n = 3$ here):

foo
bar
g
h
foo (crashed)

x
y
g
h
foo (crashed)

Same top three frames → same **bucket**, although the two inputs reached g by different routes.

- Any such rule both **over**- and **under**-counts: one bug reached through many paths, or two bugs sharing a frame. AFL's "unique crashes" (by coverage) is worse.
- Better: ask about the **cause** – crashes that one small **patch** fixes together are one bug.[24]

[24] **[ASE'18]** R. van Tonder, J. Kotheimer, C. Le Goues. "*Semantic Crash Bucketing.*"

Summary

PLRG

1. Random Testing (RT)
   - Probabilistic Analysis
   - Limits of Random Testing
2. Feedback-Directed Random Testing
3. Adaptive Random Testing (ART)
   - Distance Metrics
   - Effectiveness and Cost
   - Quasi-Random Testing
4. Fuzz Testing
   - Generation-Based Fuzzing
   - Mutation-Based Fuzzing
   - Coverage-Guided Fuzzing
   - Sanitizers and Triage

Next Lecture

- Coverage Criteria

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr