

Lecture 2 – Random Testing and Fuzzing

AAA705: Software Testing and Quality Assurance

Jihyeok Park



2026 Fall

- 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**.

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.

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

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

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

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

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

failure rate $p = \frac{|F|}{|S|}$ – 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: $p = 2^{31}/2^{32} = 0.5$.

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

failure rate $p = \frac{|F|}{|S|}$ – 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?

- Developers test the inputs they **thought of**. Random testing has no such bias.

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."

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

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.*”

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**.

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.”

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.”

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

- 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**:

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

- 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?

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

How Many Inputs Until the First Failure?

$$\begin{aligned} E(X) &= \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k (1-p)^{k-1} p \\ &= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \sum_{k=3}^{\infty} (1-p)^{k-1} + \dots \right) \end{aligned}$$

How Many Inputs Until the First Failure?

$$\begin{aligned} E(X) &= \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k (1-p)^{k-1} p \\ &= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \sum_{k=3}^{\infty} (1-p)^{k-1} + \dots \right) \\ &= p \left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \dots \right) \end{aligned}$$

How Many Inputs Until the First Failure?

$$\begin{aligned} E(X) &= \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k (1-p)^{k-1} p \\ &= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \sum_{k=3}^{\infty} (1-p)^{k-1} + \dots \right) \\ &= p \left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \dots \right) \\ &= 1 + (1-p) + (1-p)^2 + \dots \end{aligned}$$

How Many Inputs Until the First Failure?

$$\begin{aligned} E(X) &= \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k (1-p)^{k-1} p \\ &= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \sum_{k=3}^{\infty} (1-p)^{k-1} + \dots \right) \\ &= p \left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \dots \right) \\ &= 1 + (1-p) + (1-p)^2 + \dots = \boxed{\frac{1}{p}} \end{aligned}$$

How Many Inputs Until the First Failure?

$$\begin{aligned} E(X) &= \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k (1-p)^{k-1} p \\ &= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \sum_{k=3}^{\infty} (1-p)^{k-1} + \dots \right) \\ &= p \left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \dots \right) \\ &= 1 + (1-p) + (1-p)^2 + \dots = \boxed{\frac{1}{p}} \end{aligned}$$

	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

How Many Inputs Until the First Failure?

$$\begin{aligned} E(X) &= \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k (1-p)^{k-1} p \\ &= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \sum_{k=3}^{\infty} (1-p)^{k-1} + \dots \right) \\ &= p \left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \dots \right) \\ &= 1 + (1-p) + (1-p)^2 + \dots = \boxed{\frac{1}{p}} \end{aligned}$$

	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

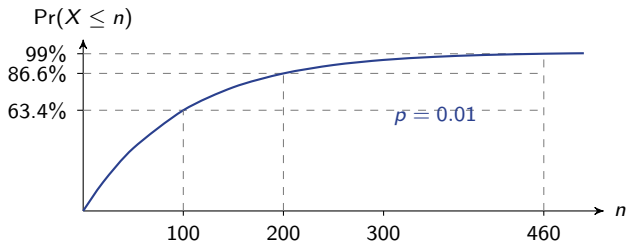
The standard deviation is as large as the mean: one run may need 10 tests, the next 300. “100 on average” is a weak promise.

How Likely Is at Least One Failure in n Inputs?

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

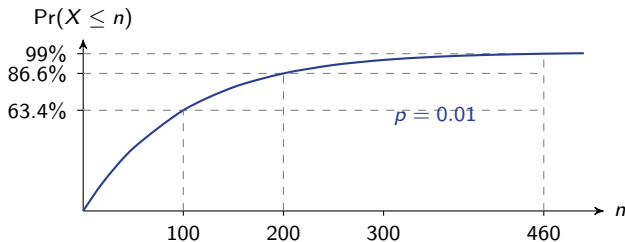
How Likely Is at Least One Failure in n Inputs?

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



How Likely Is at Least One Failure in n Inputs?

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



- 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 $1 - (1-p)^n \geq c$ for n :

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

– 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 $p = 0$?

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?

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

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**, $p < 0.5\%$.

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.

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

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

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

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

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

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}

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

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

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

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** $p = 0$.

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

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

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

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. Wüstholtz. “Estimating Residual Risk in Greybox Fuzzing.”

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

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

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

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

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

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

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

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

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

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

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
```

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.

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:

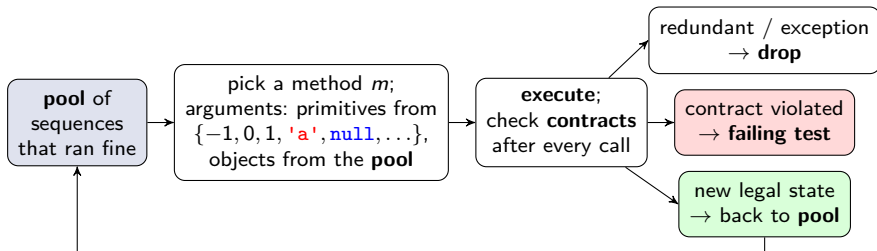
```
LinkedList l = new LinkedList();  
l.addFirst(new Object());  
TreeSet    t = new TreeSet(l);           // <- the method under test
```

- Plain random generation picks methods and arguments **blindly**. Two things go wrong:

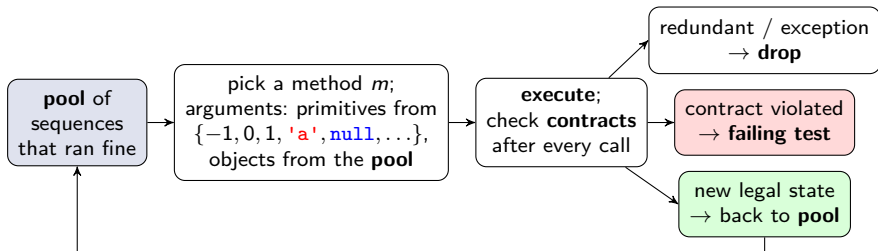
- **Illegal prefixes**. After a call that already misbehaved, every extension is wasted:

```
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.

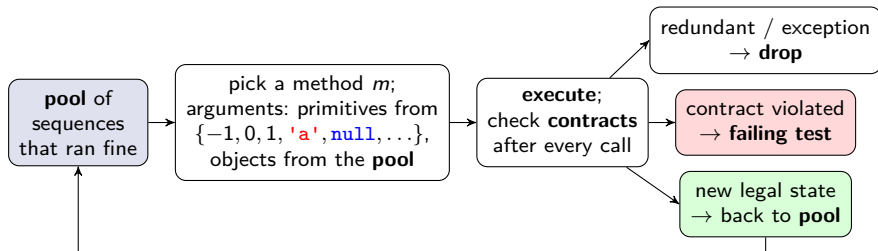


⁵[ICSE'07] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. "Feedback-Directed Random Test Generation."



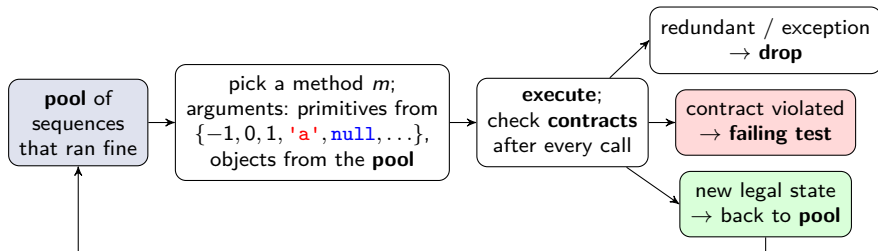
- **Build** one sequence at a time: a random method, arguments from a small pool of primitives and from objects **earlier sequences** produced.

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



- **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.

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



- **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."

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)
```

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**.

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.

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

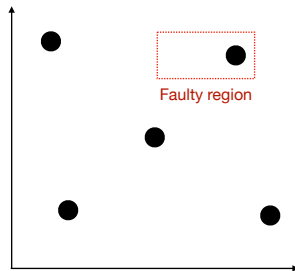
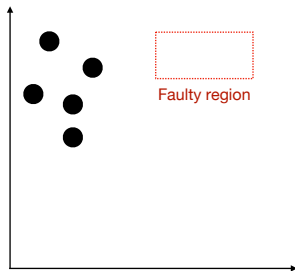
4. Fuzz Testing

Generation-Based Fuzzing

Mutation-Based Fuzzing

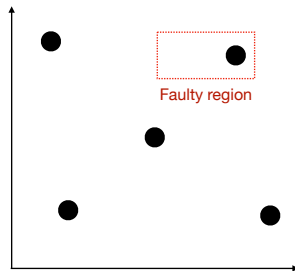
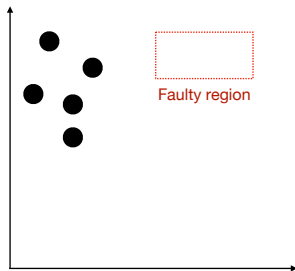
Coverage-Guided Fuzzing

Sanitizers and Triage



- A fault guarded by $x \geq 0 \ \&\& \ x < 100$ fails on a **contiguous** region – a **faulty region**. Empirically, failing inputs form **blocks**, **strips**, or scattered **points**.⁶

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



- A fault guarded by $x \geq 0 \ \&\& \ x < 100$ 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."

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:

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

- 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.

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

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

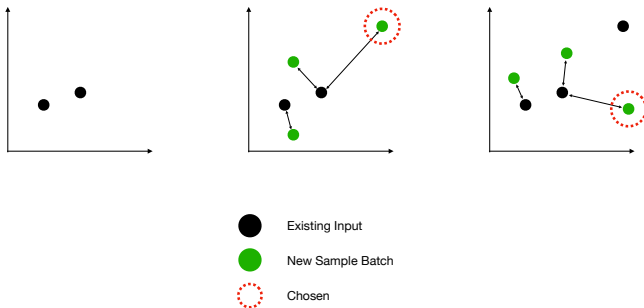
- ① $T \leftarrow \{ \text{one random input} \}$
- ② **Repeat** until the stopping criterion (budget, or a failure):
 - ① Draw Z **candidates** at random – $Z = 10$ in the original.
 - ② For each candidate c , find the **nearest** executed test and take that distance:

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

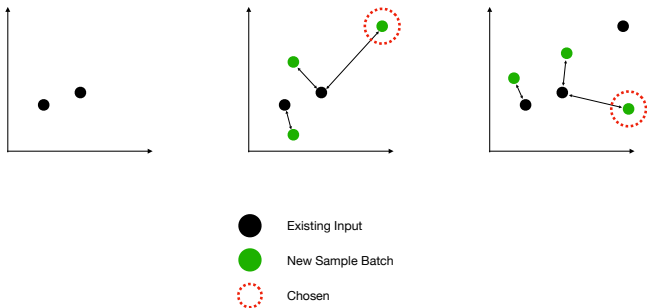
- ③ **Execute** the candidate with the **largest** fitness – the one farthest from everything tried so far – and add it to T . The other $Z - 1$ candidates are thrown away.

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

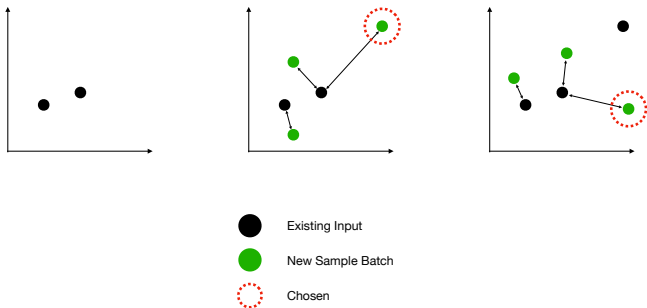
⁷[ASIAN'04] T. Y. Chen, H. Leung, I. K. Mak. "Adaptive Random Testing."



- Black: tests already executed. Green: $Z = 3$ fresh random candidates.



- 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.



- 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.

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**:

“**k**itten” $\xrightarrow[k \rightarrow s]{\text{substitute}}$ “sitt**e**n” $\xrightarrow[e \rightarrow i]{\text{substitute}}$ “sittin” $\xrightarrow[g]{\text{insert}}$ “sitting” $d = 3$

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.”

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**:

“**k**itten” $\xrightarrow[k \rightarrow s]{\text{substitute}}$ “sitt**e**n” $\xrightarrow[e \rightarrow i]{\text{substitute}}$ “sittin” $\xrightarrow[g]{\text{insert}}$ “sitting” $d = 3$

		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

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.”

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**:

“**k**itten” $\xrightarrow[k \rightarrow s]{\text{substitute}}$ “sitt**e**n” $\xrightarrow[e \rightarrow i]{\text{substitute}}$ “sittin” $\xrightarrow[g]{\text{insert}}$ “sitting” $d = 3$

		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: ARTOO⁹ 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.”

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

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."

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

- Chen & Merkel¹⁰ – **no** strategy beats random testing's F-measure by more than **50%**. ART can at best **halve** the number of tests.

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."

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

- Chen & Merkel¹⁰ – **no** strategy beats random testing's F-measure by more than **50%**. ART can at best **halve** the number of tests.
- Arcuri & Briand¹¹ – real failure rates are tiny, so k is huge, so $O(k^2Z)$ eats the gain: by **wall-clock**, plain random testing often wins.

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."

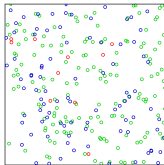
cost	$Z(0 + 1 + \dots + (k - 1)) = \frac{k(k-1)}{2}Z \implies O(k^2Z)$ distance computations
gain	F-measure (tests to first failure): $1/p$ for RT $\implies$ at best $1/2p$ for any strategy

- Chen & Merkel¹⁰ – **no** strategy beats random testing's F-measure by more than **50%**. ART can at best **halve** the number of tests.
- Arcuri & Briand¹¹ – 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.

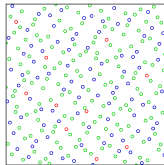
¹⁰[TOSEM'08] T. Y. Chen, R. Merkel. "An Upper Bound on Software Testing Effectiveness."

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."



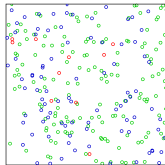
pseudo-random



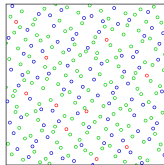
quasi-random

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

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



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:

$$\begin{aligned} & \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \dots & \frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{4}{9}, \frac{7}{9}, \frac{2}{9}, \frac{5}{9}, \dots \\ & \left(\frac{1}{2}, \frac{1}{3}\right), \left(\frac{1}{4}, \frac{2}{3}\right), \left(\frac{3}{4}, \frac{1}{9}\right), \left(\frac{1}{8}, \frac{4}{9}\right), \left(\frac{5}{8}, \frac{7}{9}\right), \left(\frac{3}{8}, \frac{2}{9}\right), \dots \end{aligned}$$

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

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

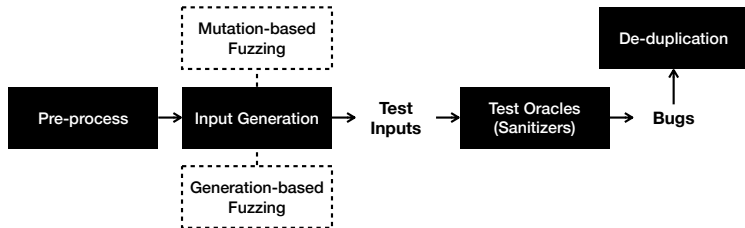
4. Fuzz Testing

Generation-Based Fuzzing

Mutation-Based Fuzzing

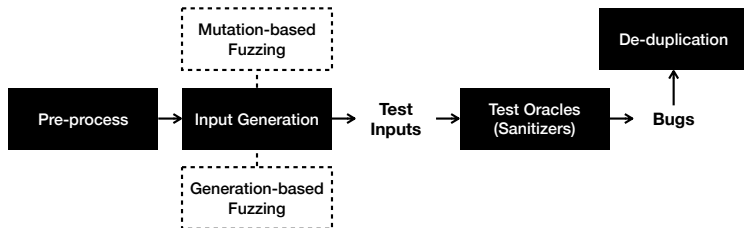
Coverage-Guided Fuzzing

Sanitizers and Triage



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

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."



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."

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
```

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
```

A grammar for arithmetic expressions:

$E ::= E + E \mid E * E \mid (E) \mid N$	$N ::= 0 \mid 1 \mid \dots \mid 9$
--	------------------------------------

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

A grammar for arithmetic expressions:

$E ::= E + E \mid E * E \mid (E) \mid N$	$N ::= 0 \mid 1 \mid \dots \mid 9$
--	------------------------------------

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

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

A grammar for arithmetic expressions:

$E ::= E + E \mid E * E \mid (E) \mid N$	$N ::= 0 \mid 1 \mid \dots \mid 9$
--	------------------------------------

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

- 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
```

¹⁵[\[NDSS'19\]](#) H. Han, D. Oh, S. K. Cha. “CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.”

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
```

CodeAlchemist¹⁵ 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
```

¹⁵[\[NDSS'19\]](#) H. Han, D. Oh, S. K. Cha. “CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.”

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
```

CodeAlchemist¹⁵ 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.

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

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
```

CodeAlchemist¹⁵ 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.”

For a C compiler a grammar is not enough: a random program with **undefined behavior** proves nothing – any output is legal. Csmith¹⁶ 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;  
}
```

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

For a C compiler a grammar is not enough: a random program with **undefined behavior** proves nothing – any output is legal. Csmith¹⁶ 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.

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

For a C compiler a grammar is not enough: a random program with **undefined behavior** proves nothing – any output is legal. Csmith¹⁶ 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."

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%)
---------------	-------	--	---------	-------	--	---------	------	--	-----	-------

¹⁷[S&P'17] J. Wang, B. Chen, L. Wei, Y. Liu. "Skyfire: Data-Driven Seed Generation for 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 $\rightarrow$ inputs that look like **real** ones, so they pass the checks real inputs pass.

¹⁷[S&P'17] J. Wang, B. Chen, L. Wei, Y. Liu. "Skyfire: Data-Driven Seed Generation for 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 $\rightarrow$ inputs that look like **real** ones, so they pass the checks real inputs pass.
- **Invert** the weights $\rightarrow$ inputs made of the **rare** productions – the corners nobody exercised (here: parentheses).

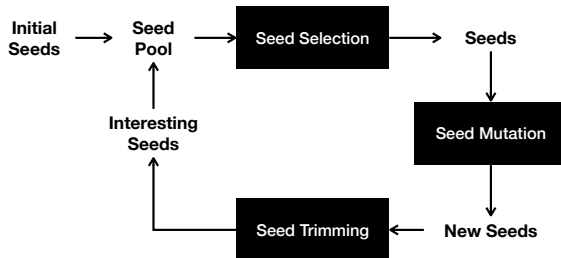
¹⁷[S&P'17] J. Wang, B. Chen, L. Wei, Y. Liu. "Skyfire: Data-Driven Seed Generation for Fuzzing."

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

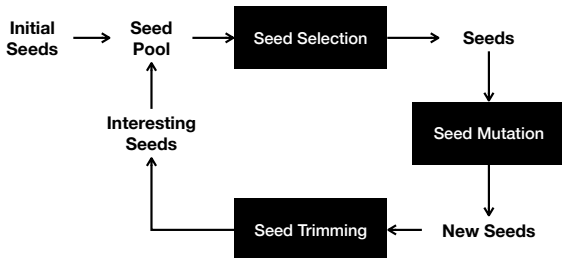
$E ::= E + E$	(45%)		$E * E$	(30%)		(E)	(5%)		N	(20%)
---------------	-------	--	---------	-------	--	---------	------	--	-----	-------

- Sample with these weights $\rightarrow$ inputs that look like **real** ones, so they pass the checks real inputs pass.
- **Invert** the weights $\rightarrow$ inputs made of the **rare** productions – the corners nobody exercised (here: parentheses).
- Skyfire¹⁷ 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."



- No model. Start from **seeds** – inputs the program accepts. Pick one, **mutate** it a little, run it, repeat.



- 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.

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

2024-06-15

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

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.

Byte mutation of **source code** mostly produces syntax errors. LangFuzz¹⁸ mutates at the level of the **grammar** instead. Two tests from the engine's test suite:

```
// test A
var a = [1, 2, 3];
a.length = 0;
```

```
// test B
var o = {};
o.__proto__ = null;
```

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

Byte mutation of **source code** mostly produces syntax errors. LangFuzz¹⁸ mutates at the level of the **grammar** instead. Two tests from the engine's test suite:

```
// test A
var a = [1, 2, 3];
a.length = 0;
```

```
// test B
var o = {};
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
```

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

Byte mutation of **source code** mostly produces syntax errors. LangFuzz¹⁸ mutates at the level of the **grammar** instead. Two tests from the engine's test suite:

```
// test A
var a = [1, 2, 3];
a.length = 0;
```

```
// test B
var o = {};
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."

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
```

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

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
```

DIE¹⁹ 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
```

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

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
```

DIE¹⁹ 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**. $2.4\times$ fewer runtime errors, $5.7\times$ 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."

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();
```

²⁰[2013] M. Zalewski. “American Fuzzy Lop.” [WOOT’20] A. Fioraldi, D. Maier, H. EiBfeldt, M. Heuse. “AFL++: Combining Incremental Steps of Fuzzing Research.”

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.

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

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.

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

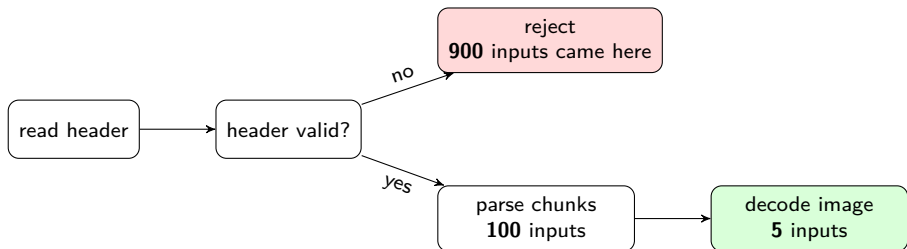
Which mutants are worth keeping as new seeds? **Measure coverage**: keep a mutant if it ran code that **no earlier input** ran.²⁰

```
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.

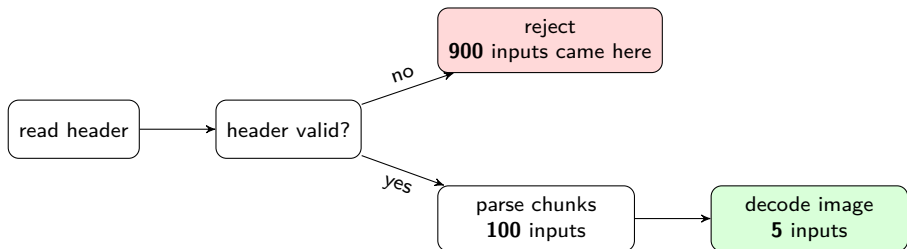
²⁰[\[2013\]](#) M. Zalewski. “American Fuzzy Lop.” [\[WOOT'20\]](#) A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse. “AFL++: Combining Incremental Steps of Fuzzing Research.”

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



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

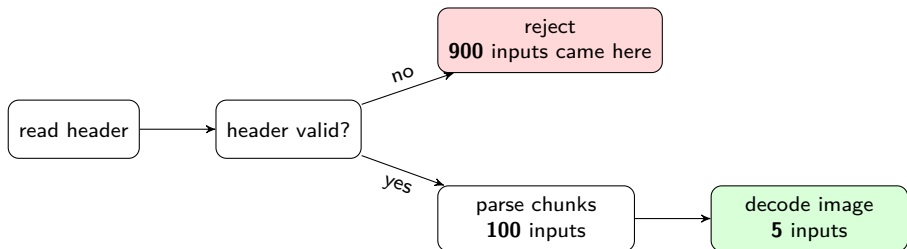
Coverage decides **which** mutants to keep, not **how many** mutations each seed gets. Count where the inputs so far have gone:



- Most seeds sit on the **common** path (900 of 1,005). Picking seeds uniformly means mutating the **rejected** inputs again and again.

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

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

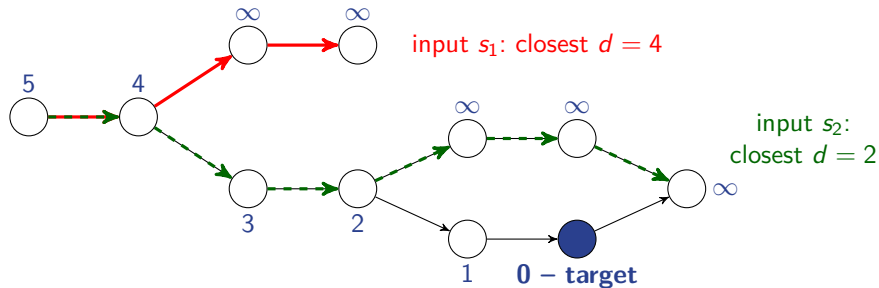


- Most seeds sit on the **common** path (900 of 1,005). Picking seeds uniformly means mutating the **rejected** inputs again and again.
- **Rarity**: a seed on a path that **few** inputs reached gets **more** mutations – its neighborhood is the least explored. AFLFast's power schedule²¹ sets energy inversely 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."

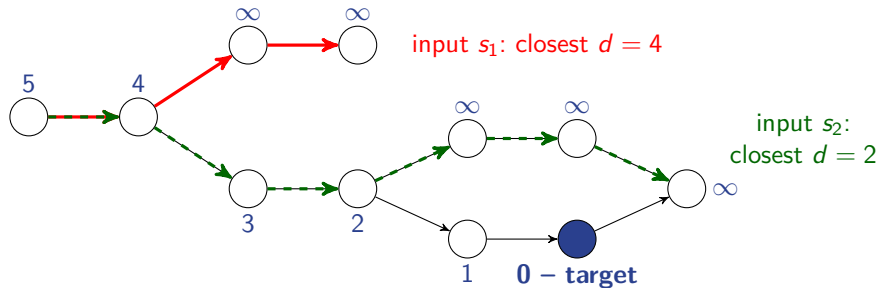
Beyond Coverage 2 – Distance (Directed Fuzzing)

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:



Beyond Coverage 2 – Distance (Directed Fuzzing)

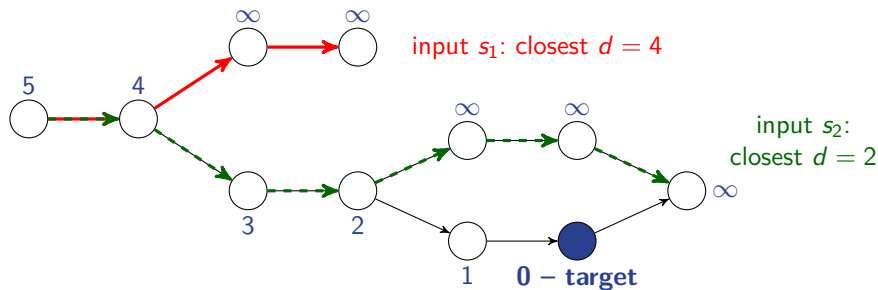
Sometimes we know **where** the bug should be – a patched line, a function in a crash report. Mark it as the **target** on the control-flow graph:



- Each node: its **shortest-path distance** d to the target, computed once. An input's distance: the **smallest** d it reached.

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.

AFLGo²² 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	$\times 27$	$\times 1$	$\times 1/27$

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

AFLGo²² 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.

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

AFLGo²² 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	×1	×1	×1
later	×27	×1	×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:

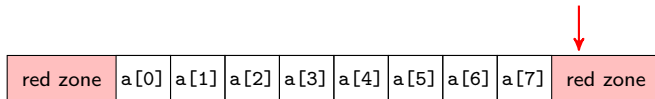
```
int *a = malloc(8 * sizeof(int));  
a[8] = 1;           // one past the end -- no crash
```

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

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
```

a[8] = 1 hits a red zone → crash

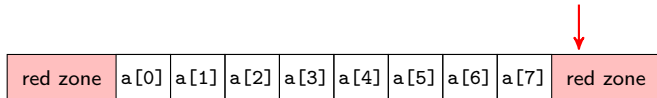


²³[USENIX ATC'12] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov. "AddressSanitizer: A Fast Address Sanity Checker."

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
```

a[8] = 1 hits a red zone → **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	x
bar	y
g	g
h	h
foo (crashed)	foo (crashed)

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

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

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	x
bar	y
g	g
h	h
foo (crashed)	foo (crashed)

Same top three frames $\rightarrow$ 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.

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

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	x
bar	y
g	g
h	h
foo (crashed)	foo (crashed)

Same top three frames $\rightarrow$ 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."

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

- Coverage Criteria

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