

# 软件分析与验证前沿

苏亭

软件科学与技术系

# Random (Fuzz) Testing

# Random (Fuzz) Testing

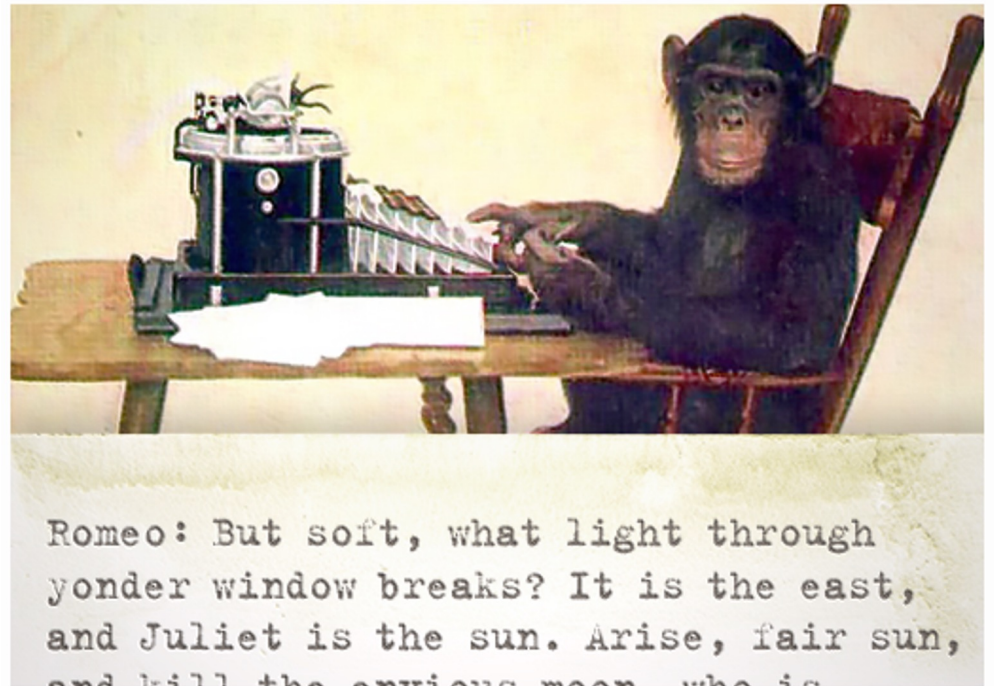
---

- Feed random inputs to a program
- Observe whether it behaves “correctly”
  - Execution satisfies given specification
  - Or just doesn’t crash
    - A simple specification
- Special case of mutation analysis

# The Infinite Monkey Theorem

---

“A monkey hitting keys at random on a typewriter keyboard will produce any given text, such as the complete works of Shakespeare, with probability approaching 1 as time increases.”



# Random Testing: Case Studies

---

- UNIX utilities: Univ. of Wisconsin's Fuzz study
  - An Empirical Study of the Reliability of UNIX Utilities  
<http://www.paradyn.org/papers/fuzz.pdf>
  - Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services <http://www.paradyn.org/papers/fuzz-revisited.pdf>
- C/C++ Programs: Greybox fuzzing in AFL
  - <https://afl-1.readthedocs.io/en/latest/index.html>
- Mobile apps: Google's Monkey tool for Android
  - <https://developer.android.com/studio/test/other-testing-tools/monkey>

# Random Testing: Case Studies

---

- UNIX utilities: Univ. of Wisconsin's Fuzz study
  - An Empirical Study of the Reliability of UNIX Utilities  
<http://www.paradyn.org/papers/fuzz.pdf>
  - Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services <http://www.paradyn.org/papers/fuzz-revisited.pdf>
- C/C++ Programs: Greybox fuzzing in AFL
  - <https://afl-1.readthedocs.io/en/latest/index.html>
- Mobile apps: Google's Monkey tool for Android
  - <https://developer.android.com/studio/test/other-testing-tools/monkey>

# The First Fuzzing Study

---

- Conducted by Barton Miller @ Univ of Wisconsin  
<https://pages.cs.wisc.edu/~bart/fuzz/fuzz.html>
- **1990**: Command-line fuzzer, testing reliability of UNIX programs
  - Bombards utilities with random data
- **1995**: Expanded to GUI-based programs (X Windows), network protocols, and system library APIs
- **Later**: Command-line and GUI-based Windows and OS X apps

# Fuzzing UNIX Utilities: Aftermath

---

- **1990:** Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely)
- **1995:** Systems got better... but not by much!

“Even worse is that many of the same bugs that we reported in 1990 are still present in the code releases of 1995.”



# Fuzzing UNIX Utilities: Aftermath

---

- **1990:** Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely)
- **1995:** Systems got better... but not by much!
- **2020:** After more than thirty years, it appears that there is still a place for basic fuzz testing.

# A Silver Lining: Security Bugs

---

- `gets()` function in C has no parameter limiting input length
  - ⇒ programmer must make assumptions about structure of input
- Causes reliability issues and security breaches
  - Second most common cause of errors in 1995 study
- Solution: Use `fgets()`, which includes an argument limiting the maximum length of input data

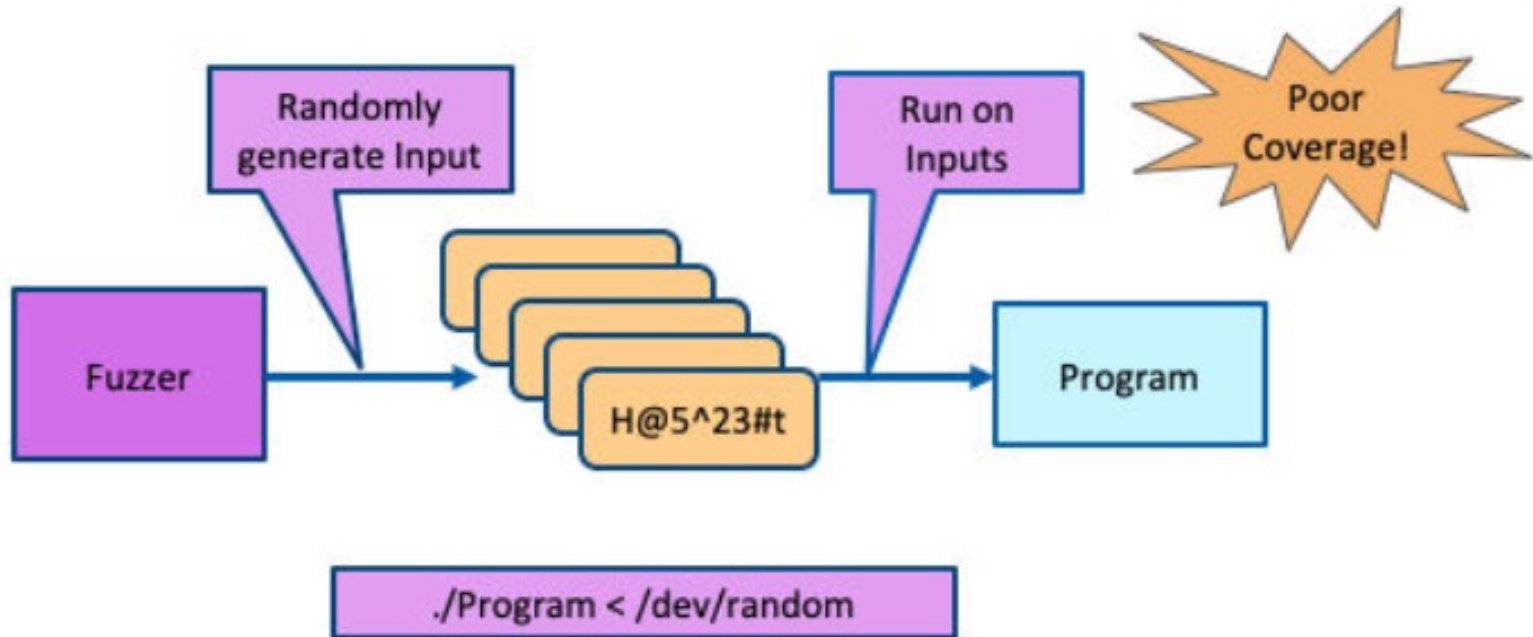
Utility	Linux		MacOS		FreeBSD		Utility	Linux		MacOS		FreeBSD	
	version	fail	version	fail	version	fail		version	fail	version	fail	version	fail
as	2.30	o	11.0.0		2.17.50	o	look	*		1.18.10		8.2	o
awk	4.1.4		20070501		20121220		m4	1.4.18		1.4.6		1.4.18_1	
bash	4.4.20		3.2.57	•	5.0.16		mail	*		8.2		8.2	
bc	1.07.1		1.07.1		1.1		make	4.1		3.8.1		8.3	•
bison	3.0.4	o	3.3	o	3.4.2		md5/md5sum	8.28		1.34		*	
calendar	*		1.19	•	8.3		mig	—		116		—	
cat	8.28		1.32		8.2		more	2.31.1		—		—	
checknr	—		1.9		8.1	•	neqn	1.22.3		1.19.2		1.19.2	
clang	8.0.0		11.0.0		8.0.0		nm	2.30		11.0.0		3504	
cmp	3.6		2.8.1		8.3		pdftex	6.2.3		6.2.3	•	6.2.1	
col	*		1.19		8.5	•	pic	1.22.3		1.19.2		1.19.2	
colcrt	*		1.18		8.1		pr	8.28		1.18		8.2	
colrm	*		1.12		8.2		ptx	8.28	o	—		—	
comm	8.28		1.21		8.4		refer	—		1.19.2		1.19.2	
compress	—		1.23		8.2		rev	2.31.1		1.12		8.3	
csd	20110502-5		—		—		sdiff	3.6		2.8.1		1.36	
ctags	25.2		5.8_1	•	8.4	•	sed	4.4		1.39		8.2	
cut	8.28		1.30		8.3		sh	—		—		8.6	•
dash	0.5.10.2-6		*		0.5.10.2		soelim	1.22.3		1.19.2		*	
dc	1.4.1	o	1.3	• o	1.3		sort	8.28		2.3		2.3	
dd	8.28		1.36		8.5		spell	1.0	o	—		—	
diff	3.6		2.8.1		2.8.7		split	8.28		1.17		8.2	
ed	1.10		*		1.5		strings	2.30		*		r3614M	
eqn	1.22.3		1.19.2		1.19.2		strip	2.30		*		r3614M	
ex/vim	8.0		8.1		8.1		sum	8.28		1.17		*	
expand	8.28		1.15		8.1		tail	8.28		101.40.1		8.1	
flex	2.6.4		2.5.35		2.5.37	•	tbl	1.22.3		1.19.2		1.19.2	
fmt	8.28		1.22		8.1		tcsh	—		6.21.00		6.20.00	
fold	8.28		1.13		8.1		tee	1.22.3		1.6		8.1	
ftp	0.17-34.1		—		8.6	•	telnet	1.14		1.16		8.4	
gcc	7.4.0		—		9.2.0		tex	6.2.3	•	6.2.3		6.2.1	
gdb	8.1.0	•	8.3.1	•	6.1.1	•	top	3.3.12		125		3.5beta12	
gfortran	7.4.0		—		—		tr	8.28		1.24		8.2	
grep	3.1		2.5.1		2.5.1		troff	1.22.3	•	1.19.2	•	1.19.2	•
grn	—		1.19.2		1.19.2		tsort	8.28		1.13		8.3	
groff	1.22.3		1.19.2	o	1.19.2	o	ul	*		101.40.1		8.1	
head	8.28		1.20		8.2		uniq	8.28		101.40.1		8.3	
htop	2.1.0		2.2.0		2.2.0		units	—		*		*	
indent	—		5.17	•	5.17	•	wc	8.28		1.21		8.1	
join	8.28		1.2		8.6		xargs	4.7.0		1.57		8.1	
less	551	•	487	•	530		zic	2.27		8.22		8.22	
lldb	—		9.0.1	•	8.0.0	•	zsh	5.4.2		5.7.1		5.7.1	•

87 utilities were tested on Unix, MacOS, and freeBSD, 67 of which were tested on all three systems.

• = crashed, o = hung, — = unavailable on that system, \* = version information unavailable.

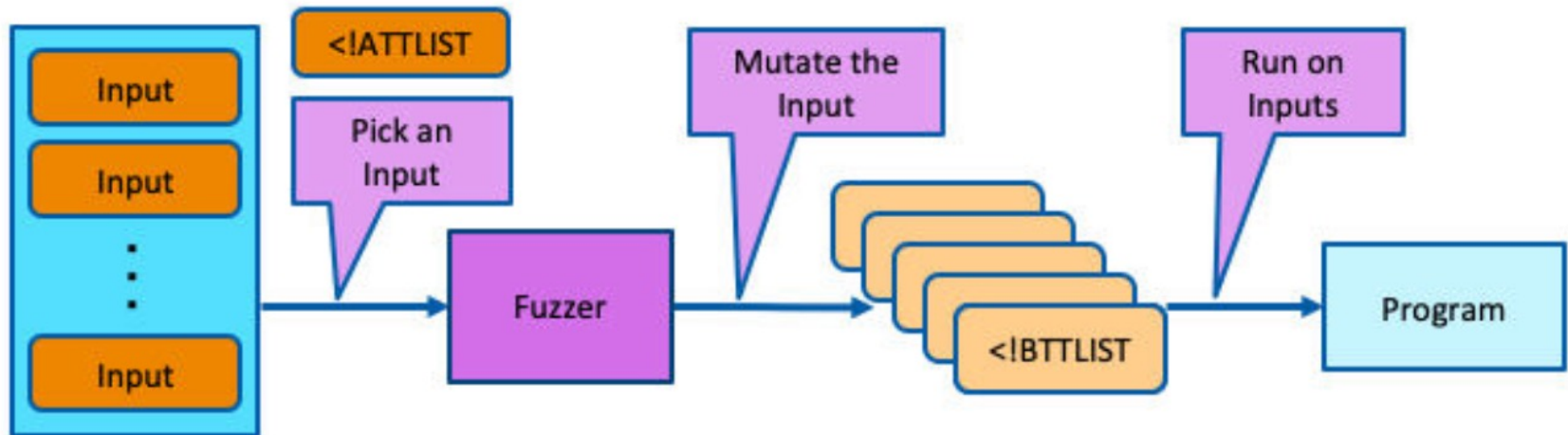
# First Generation of Fuzzers

---

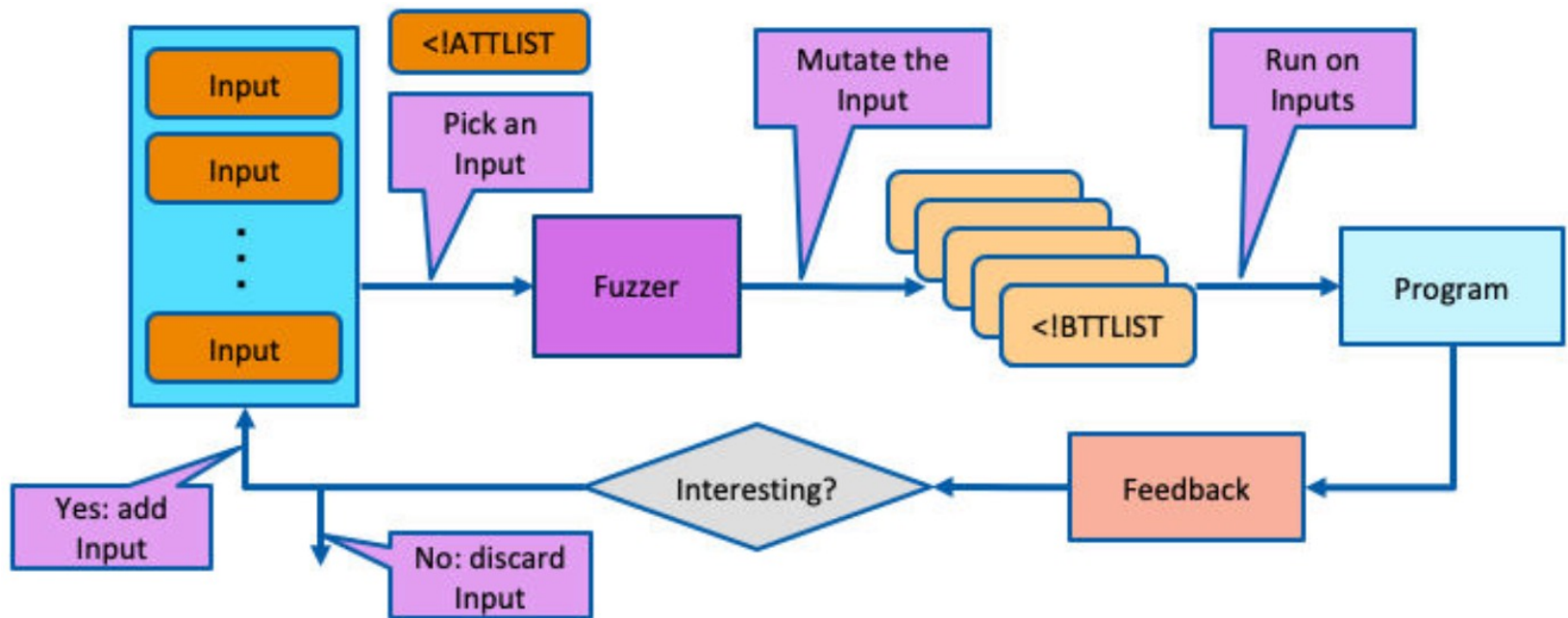


# Second Generation of Fuzzers

---



# Third Generation of Fuzzers



# What Types of Fuzzers?

---

- **Mutation-based**
  - introduce small changes to existing inputs that may still keep the input valid yet exercise new behavior
- **Grammar-based**
  - provide a *specification* of the legal inputs to a program for very systematic and efficient test generation, in particular for complex input formats
- **Search-based**
  - adopt search algorithms to reach some targets more quickly

# What Kinds of Bugs can Fuzzing Find?

---

- Memory errors
  - Spatial (e.g., out-of-bound access) and temporal (e.g., use-after-free)
- Other undefined behaviors
  - Integer overflow, divide-by-zero, null deference, uninitialized read, ...
- Assertion violations
- Infinite loops (using timeout)
- Concurrency bugs
  - Data race, deadlock, ...



# Random Testing: Pros and Cons

---

## Pros:

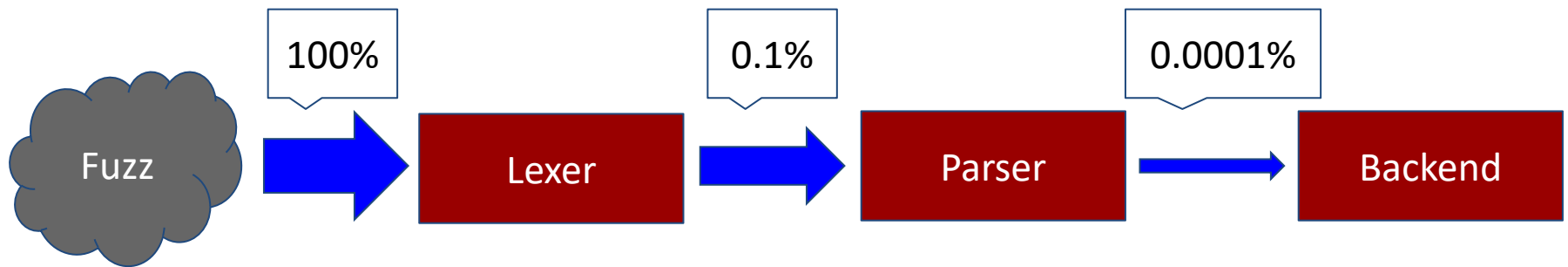
- Easy to implement
- Provably good coverage given enough tests
- Can work with programs in any format
- Appealing for finding security vulnerabilities

## Cons:

- Inefficient test suite
- Might find bugs that are unimportant
- Poor coverage

# Coverage of Random Testing

---



- The lexer is very heavily tested by random inputs
- But testing of later stages is much less efficient

# Random Testing: Case Studies

---

- UNIX utilities: Univ. of Wisconsin's Fuzz study
  - An Empirical Study of the Reliability of UNIX Utilities  
<http://www.paradyn.org/papers/fuzz.pdf>
  - Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services <http://www.paradyn.org/papers/fuzz-revisited.pdf>
- C/C++ Programs: Greybox fuzzing in AFL
  - <https://afl-1.readthedocs.io/en/latest/index.html>
- Mobile apps: Google's Monkey tool for Android
  - <https://developer.android.com/studio/test/other-testing-tools/monkey>

# Greybox Fuzzing in AFL

---

- **Guide input generation toward a goal**
  - Guidance based on **lightweight program analysis**
- **Three main steps**
  - **Randomly** generate inputs
  - Get feedback from test executions: What code is **covered**?
  - **Mutate inputs** that have covered new code

# American Fuzzy Lop

---



# American Fuzzy Lop

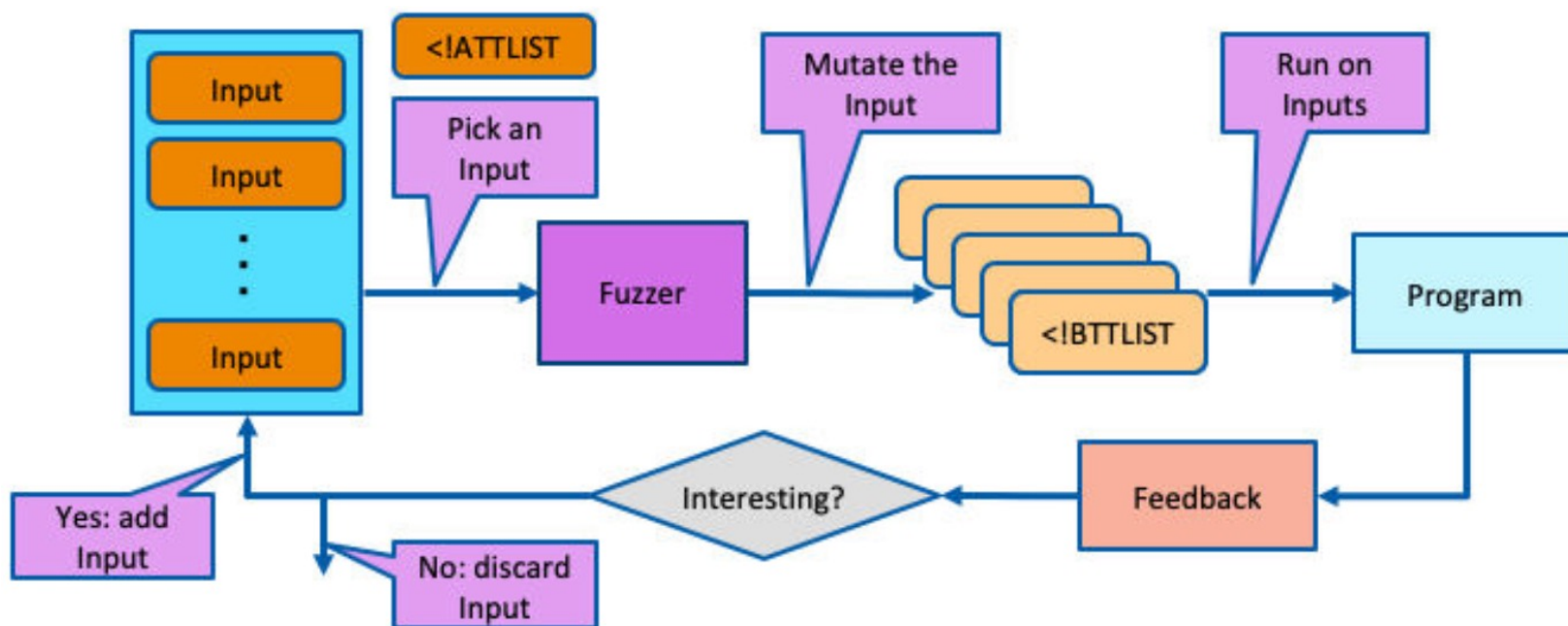
---

- **Simple yet effective** fuzzing tool
  - Targets C/C++ programs
  - Inputs are, e.g., files read by the program
- **Widely used in industry**
  - In particular, to find **security-related bugs**
  - E.g., in OpenSSL, PHP, Firefox

# Workflow of AFL

## (Third Generation of Fuzzers)

---



<https://afl-1.readthedocs.io/en/latest/index.html>

[https://afl-1.readthedocs.io/en/latest/about\\_afl.html#more-about-afl](https://afl-1.readthedocs.io/en/latest/about_afl.html#more-about-afl)

# Measuring Coverage

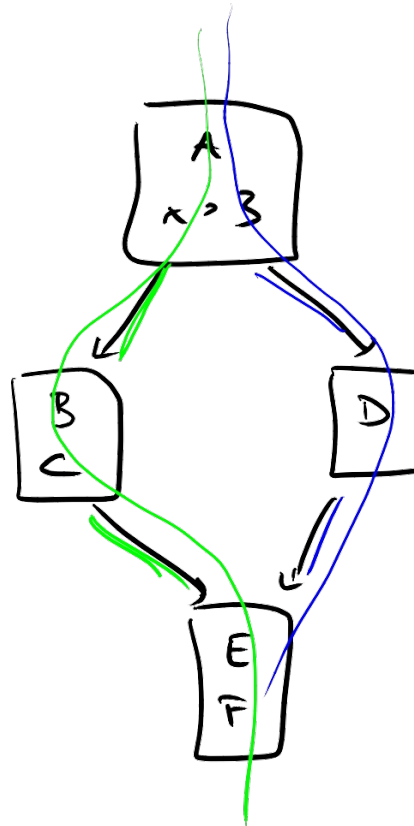
---

- **Different coverage metrics**
  - Line/statement/branch/path coverage
- **Here: Branch (edge) coverage**
  - Branches between basic blocks
    - Rationale: Reaching a code location not enough to trigger a bug, but state also matters
  - **Compromise** between
    - **Effort** spent on measuring coverage
    - **Guidance** it provides to the fuzzer



## Example

```
A
if (x > 3) {
  B
  C
} else {
  D
}
E
F
```



Exec. 1

Exec. 2

A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)

A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)

# Efficient Implementation

---

- **Instrumentation added at branching points:**

```
cur_location =/ *COMPILE_TIME_RANDOM*/;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

# Efficient Implementation

---

- Instrumentation added at branching points:

```
cur_location = / *COMPILE TIME RANDOM* /;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

**Advantage:**  
**Works well with separate  
compilation**

# Efficient Implementation

---

- **Instrumentation added at branching points:**

```
cur_location =/ *COMPILE_TIME_RANDOM*/;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

**Globally reachable memory  
location that stores how often  
each edge was covered**

# Efficient Implementation

---

- **Instrumentation added at branching points:**

```
cur_location =/ *COMPILE_TIME_RANDOM*/;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

**a 64 kB SHM region**

- (1) *Large enough* to ensure that collisions are sporadic with almost all of the intended targets (2k~10k discoverable branch points).
- (2) *Small enough* to allow the map to be analyzed in microseconds on the receiving end, and to effortlessly fit within L2 cache.

# Efficient Implementation

---

- Instrumentation added at branching points:

```
cur_location =/ *COMPILE_TIME_RANDOM*/;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```


Combine previous and current  
block into a fixed-size hash

# Efficient Implementation

---

- Instrumentation added at branching points:

```
cur_location =/ *COMPILE_TIME_RANDOM*/;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

  
Shift to distinguish between “A” followed by  
“B” from “B” followed by “A”

# Detecting New Behaviors

---

- Inputs that **trigger a new edge** in the CFG: Considered as **new behavior**
- **Alternative: Consider new paths**
  - More expensive to track
  - Path explosion problem

#1: A -> B -> C -> D -> E

#2: A -> B -> C -> A -> E *new*

#3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E *not new*



# Edge Hit Counts

---

- **Refinement of the previous definition of “new behaviors”**
- **For each edge, count how often it is taken**
  - Approximate counts based on buckets of increasing size
    - 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+.
  - Rationale: Focus on relevant differences in the hit counts

# Evolving the Input Queue

---

- **Maintain queue of inputs**
  - Q Initially: **Seed inputs** provided by user
  - Q Once used, **keep** input if it **covers new edges**
  - Q **Add new inputs by mutating existing input**
- **In practice: Queue sizes of 1k to 10k**

# Mutation Operators

---

- **Goal: Create new inputs from existing inputs**
- **Random transformations** of bytes in an existing input
  - Q **Bit flips** with varying lengths and stepovers
  - Q **Addition and subtraction** of small integers
  - Q **Insertion** of known interesting integers
    - E.g., 0, 1, INT-MAX
  - Q **Splicing** of different inputs

# More Tricks for Fast Fuzzing

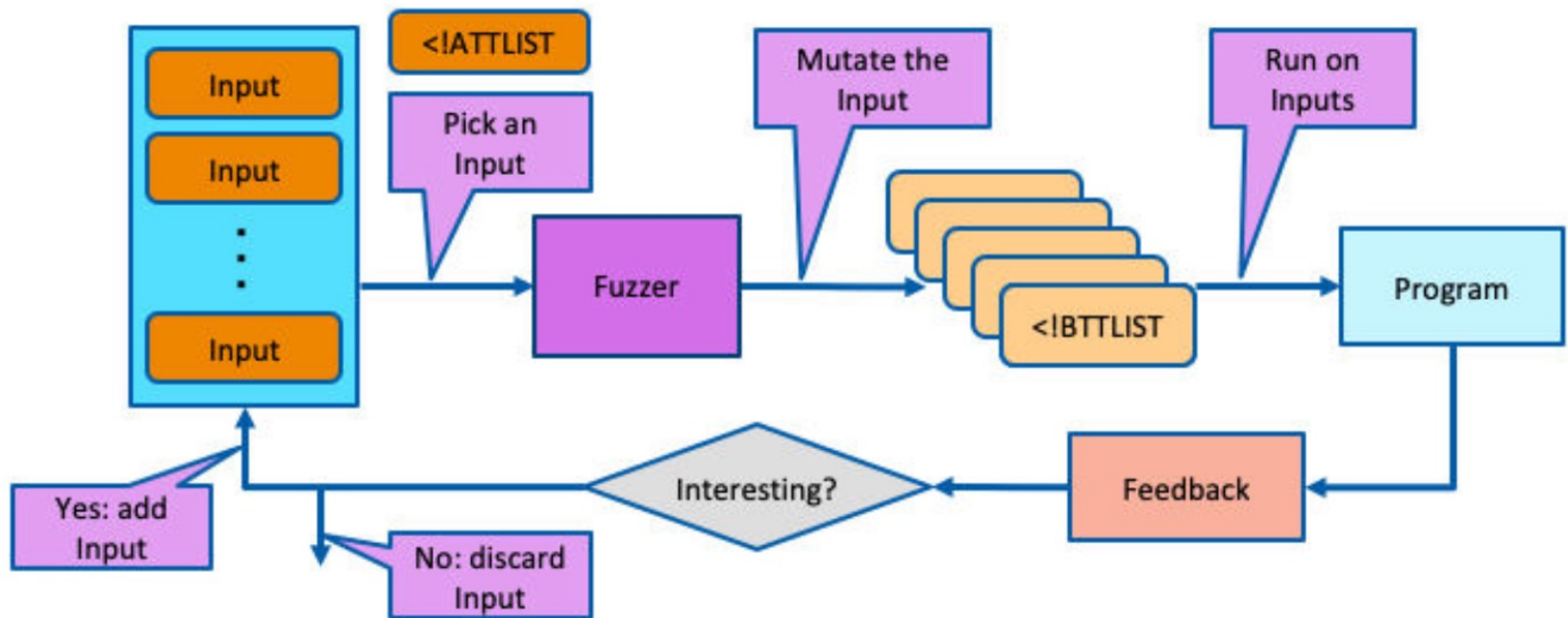
---

- **Time and memory limits**
  - Q Discard input when execution is too expensive
- **Pruning the queue**
  - Q Periodically select subset of inputs that still cover every edge seen so far
- **Prioritize how many mutants to generate from an input in the queue**
  - Q E.g., focus on unusual paths or try to reach specific locations
- **The fork server**

# Revisit: Workflow of AFL

## (Third Generation of Fuzzers)

---



<https://afl-1.readthedocs.io/en/latest/index.html>

[https://afl-1.readthedocs.io/en/latest/about\\_afl.html#more-about-afl](https://afl-1.readthedocs.io/en/latest/about_afl.html#more-about-afl)

# Real-World Impact

---

- **Open-source tool** maintained mostly by **Google**
  - Q Initially created by single developer
  - Q Various improvements proposed in academia and industry
- **Fuzzers regularly check various security-critical components**
  - Q Many **thousands of compute hours**
  - Q **Hundreds of detected vulnerabilities**

# Random Testing: Case Studies

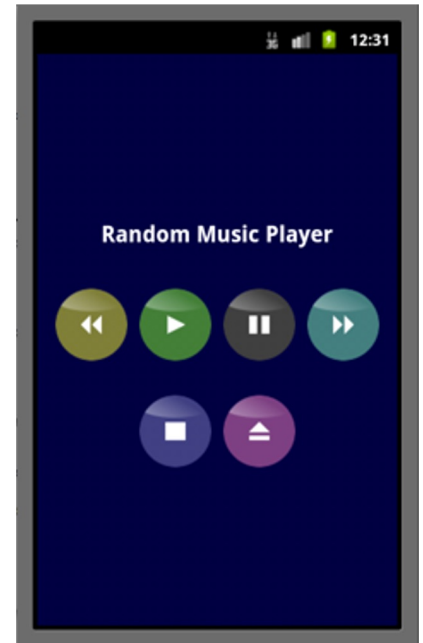
---

- UNIX utilities: Univ. of Wisconsin's Fuzz study
  - An Empirical Study of the Reliability of UNIX Utilities  
<http://www.paradyn.org/papers/fuzz.pdf>
  - Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services <http://www.paradyn.org/papers/fuzz-revisited.pdf>
- C/C++ Programs: Greybox fuzzing in AFL
  - <https://afl-1.readthedocs.io/en/latest/index.html>
- Mobile apps: Google's Monkey tool for Android
  - <https://developer.android.com/studio/test/other-testing-tools/monkey>

# Fuzz Testing for Mobile Apps

---

```
class MainActivity extends Activity implements
OnClickListener {
    void onCreate(Bundle bundle) {
        Button buttons = new Button[] { play, stop, ... };
        for (Button b : buttons) b.setOnClickListener(this);
    }
    void onClick(View target) {
        switch (target) {
            case play:
                startService(new Intent(ACTION_PLAY));
                break;
            case stop:
                startService(new Intent(ACTION_STOP));
                break;
            ...
        }
    }
}
```



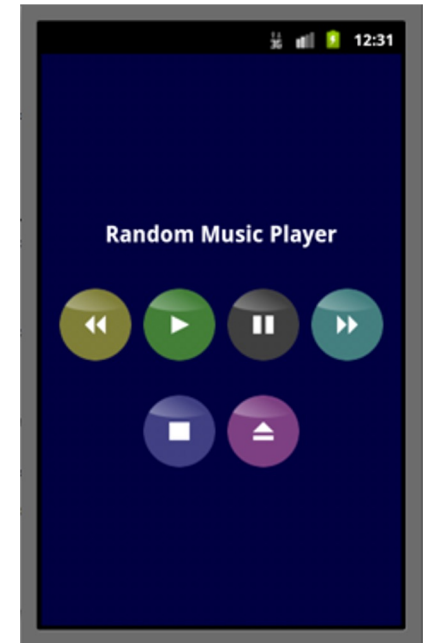


# Generating Single-Input Events

```
class MainActivity extends Activity implements
OnClickListener {
    void onCreate(Bundle bundle) {
        Button buttons = new Button[] { play, stop, ... };
        for (Button b : buttons) b.setOnClickListener(this);
    }
    void onClick(View target) {
        switch (target) {
            case play:
                startService(new Intent(ACTION_PLAY));
                break;
            case stop:
                startService(new Intent(ACTION_STOP));
                break;
            ...
        }
    }
}
```

TOUCH(136,351)

TOUCH(136,493)

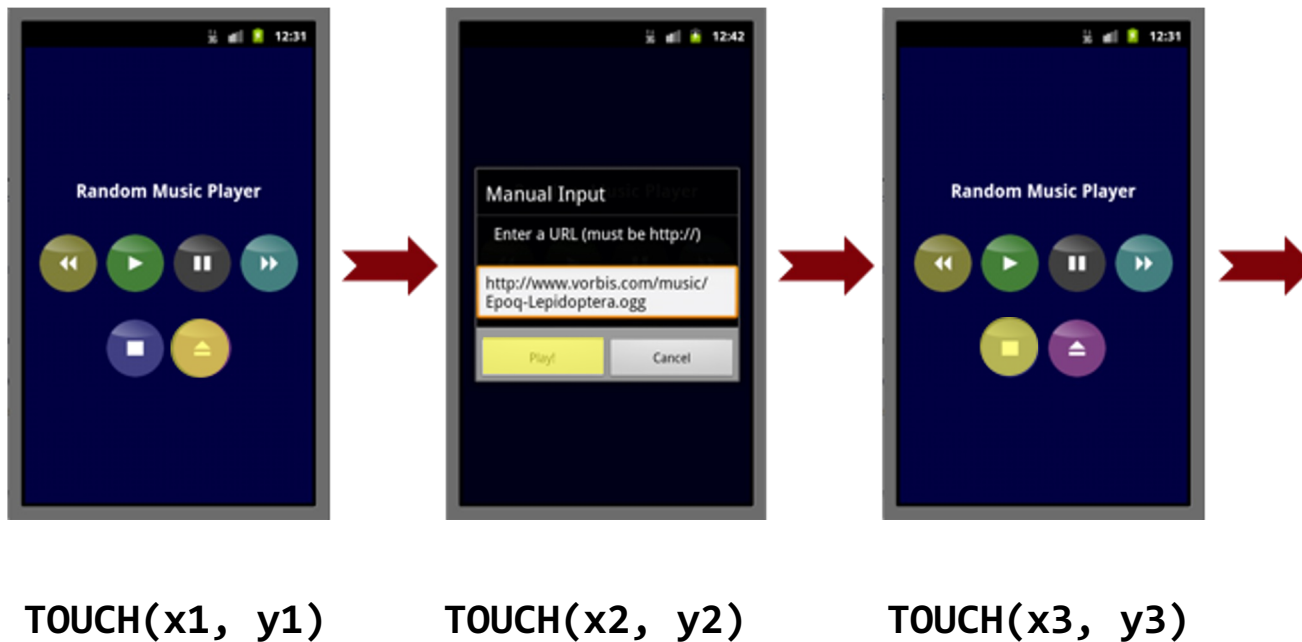


**TOUCH(x, y)** where x, y are randomly generated:

x in [0..480], y in [0..800]

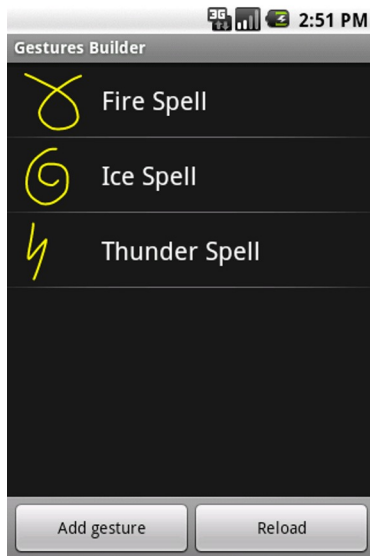
# Black-Box vs. White-Box Testing

---

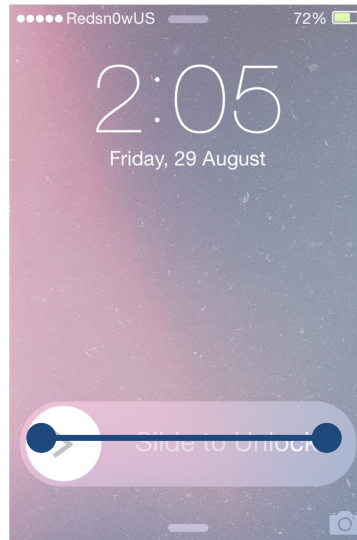


# Generating Gestures

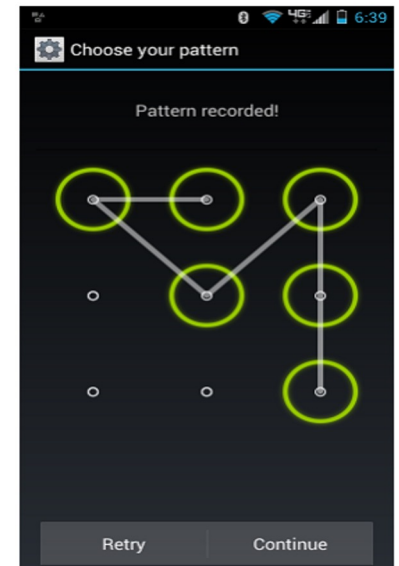
DOWN( $x_1, y_1$ ) MOVE( $x_2, y_2$ ) UP( $x_2, y_2$ )



( $x_1, y_1$ )



( $x_2, y_2$ )



# Grammar of Monkey Events

---

*test\_case* := *event* \*

*event* := *action* ( *x* , *y* ) | ...

*action* := **DOWN** | **MOVE** | **UP**

*x* := **0** | **1** | ... | *x\_limit*

*y* := **0** | **1** | ... | *y\_limit*

# Effectiveness of Monkey

---

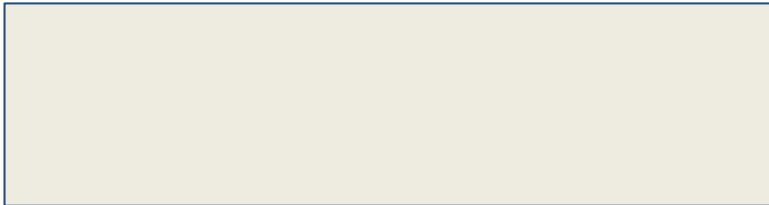
- Monkey is still one of the most effective GUI testing tool for Android.
- Industrial companies adapts and runs Monkey for daily testing.
  - FastBot (ByteDance)、 Sapienz (Facebook)
- Thousands of crashing bugs were found

# QUIZ: Monkey Events

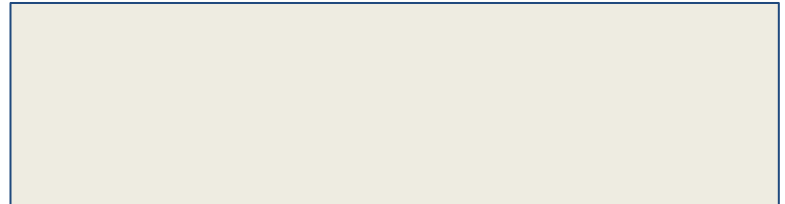
---

Give the correct specification of TOUCH and MOTION events in Monkey's grammar using UP, MOVE, and DOWN statements.

Give the specification of a TOUCH event at pixel (89,215).



Give the specification of a MOTION event from pixel (89,215) to pixel (89,103) to pixel (371,103).



# QUIZ: Monkey Events

---

Give the correct specification of TOUCH and MOTION events in Monkey's grammar using UP, MOVE, and DOWN statements.

Give the specification of a TOUCH event at pixel (89,215).

```
DOWN(89,215) UP(89,215)
```

TOUCH events are a pair of DOWN and UP events at a single place on the screen.

Give the specification of a MOTION event from pixel (89,215) to pixel (89,103) to pixel (371,103).

```
DOWN(89,215) MOVE(89,103)  
MOVE(37,103) UP(37,103)
```

MOTION events consist of a DOWN event somewhere on the screen, a sequence of MOVE events, and an UP event.

# What Have We Learned?

---

## Random testing:

- Is effective for testing security, classic programs, mobile apps, etc
- Should complement not replace systematic, formal testing
- Must generate test inputs from a reasonable distribution to be effective
- May be less effective for systems with multiple layers (e.g. compilers)



# Extended Reading

---

- UNIX utilities: Univ. of Wisconsin's Fuzz study
  - An Empirical Study of the Reliability of UNIX Utilities  
<http://www.paradyn.org/papers/fuzz.pdf>
  - Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services <http://www.paradyn.org/papers/fuzz-revisited.pdf>
- Mobile apps: Google's Monkey tool for Android
  - <https://developer.android.com/studio/test/other-testing-tools/monkey>
- C/C++ Programs: Greybox fuzzing in AFL
  - <https://afl-1.readthedocs.io/en/latest/index.html>