Carnegie Mellon University

Advanced Testing

17-313, Foundations of Software Engineering, Fall 2023

S3D
Software and Societal Systems Department

Learning Goals

- Describe random test-input generation strategies such as fuzz testing
- Characterize challenges of performance testing and suggest strategies
- Reason about failures in microservice applications how chaos engineering can be applied to test resiliency of cloud-based applications
- Describe A/B testing for usability
- Identify the need for chaos and resilience engineering, and the principles of chaos engineering

Outline

- More static analysis
  - annotations
- Fuzz Testing
- Testing Performance
- Testing Usability
- Chaos!

Java Checker Framework

Uses annotations to detect common errors

- Uses a conservative analysis to prove the absence of certain defects *
  - Null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings, …
  - C.f. SpotBugs which makes no safety guarantees
  - Assuming that code is annotated and those annotations are correct
- Uses annotations to enhance Java's type system

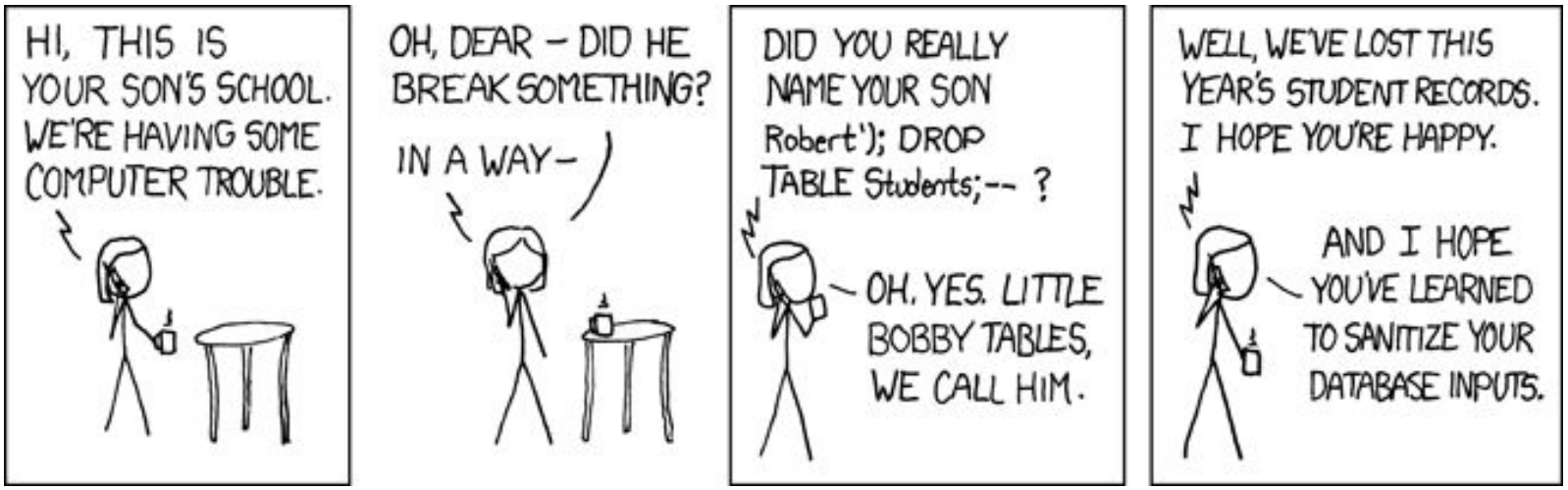
https://checkerframework.org/

Taint Analysis

Prevents untrusted (tainted) data from reaching sensitive locations (sinks)

- Tracks flow of sensitive information through the program
- Tainted inputs come from arbitrary, possibly malicious sources
  - User inputs, unvalidated data
- Using tainted inputs may have dangerous consequences
  - Program crash, data corruption, leak private data, etc.
- We need to check that inputs are sanitized before reaching sensitive locations

Classic Example: SQL Injection

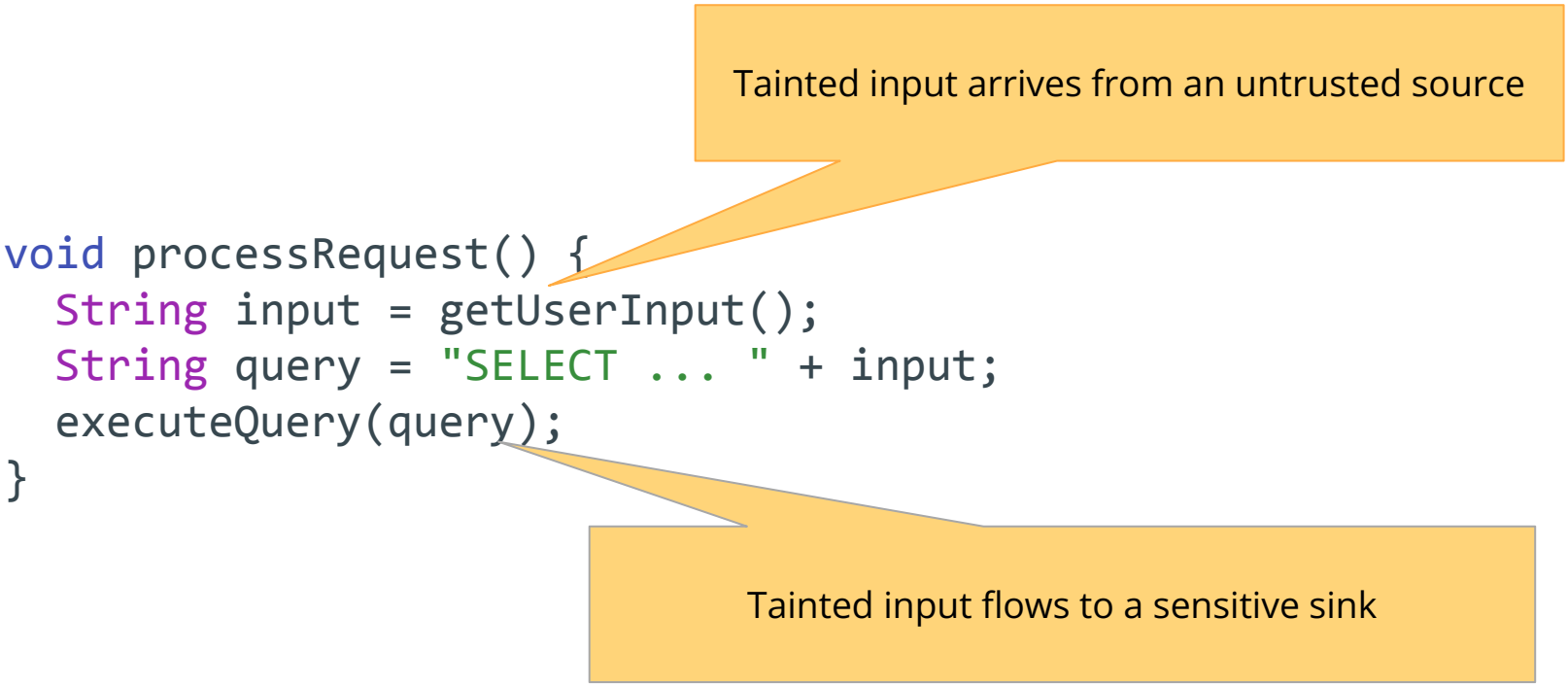


https://xkcd.com/327

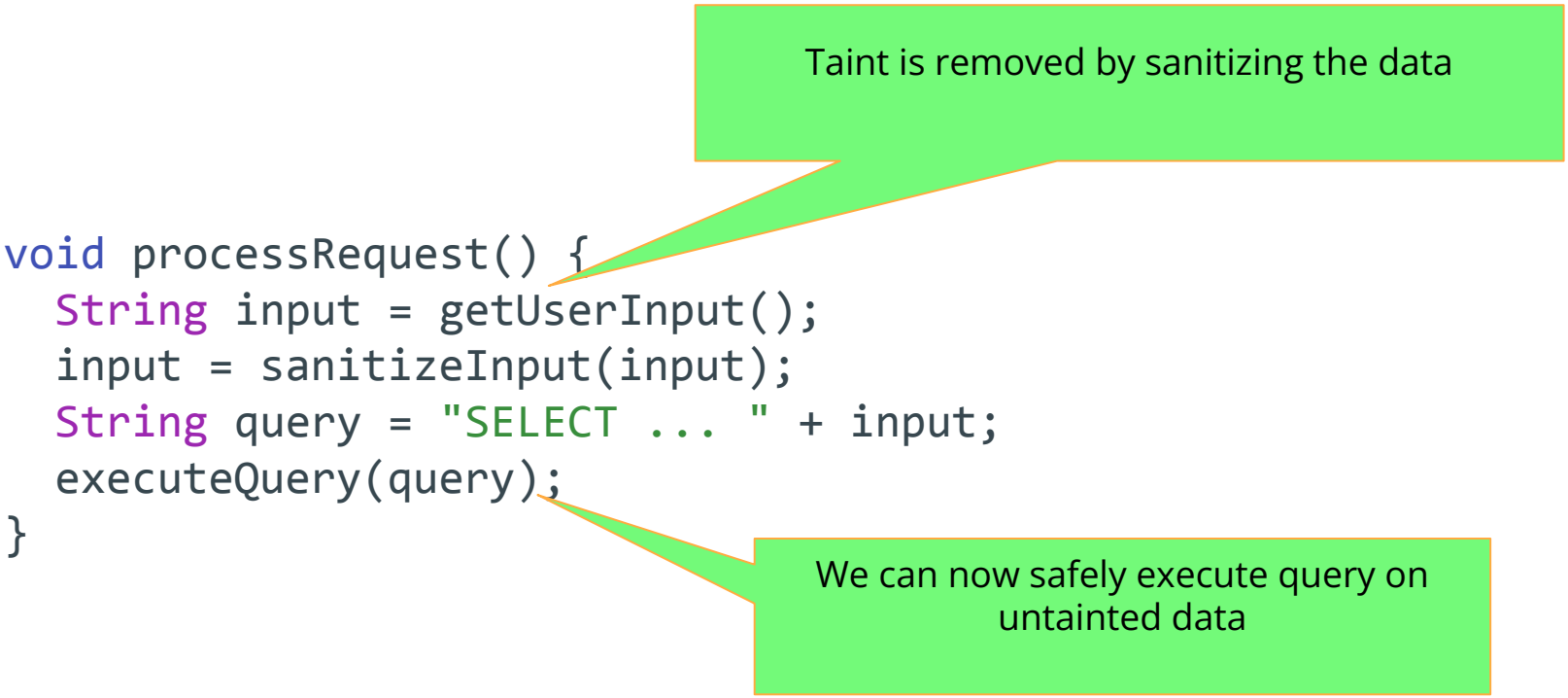
Classic Example: SQL Injection

```
void processRequest() {
  String input = getUserInput();
  String query = "SELECT ... " + input;
  executeQuery(query);
}
```

Classic Example: SQL Injection



Classic Example: SQL Injection



Taint Checker: `@Tainted` and `@Untainted`

```
void processRequest() {
  @Tainted String input = getUserInput();
  executeQuery(input);
}

public void executeQuery(@Untainted String input) {
  // ...
}

@Untainted public String validate(String userInput) {
  // ...
}
```

Taint Checker: `@Tainted` and `@Untainted`

```
void processRequest() {
  @Tainted String input = getUserInput();
  executeQuery(input);
}

public void executeQuery(@Untainted String input) {
  // ...
}

@Untainted public String validate(String userInput) {
  // ...
}
```

Indicates that data is tainted

Argument ***must*** be untainted

Guarantees that return value is untainted

Taint Checker: `@Tainted` and `@Untainted`

```
void processRequest() {
  @Tainted String input = getUserInput();
  executeQuery(input);
}

public void executeQuery(@Untainted String input) {
  // ...
}

@Untainted public String validate(String userInput) {
  // ...
}
```

Indicates that data is tainted

Argument ***must*** be untainted

Guarantees that return value is untainted

Does this compile?

```
void processRequest() {
  @Tainted String input = getUserInput();
  input = validate(input);
  executeQuery(input);
}

public void executeQuery(@Untainted String input) {
  // ...
}

@Untainted public String validate(String userInput) {
  // ...
}
```

Input becomes @Untainted

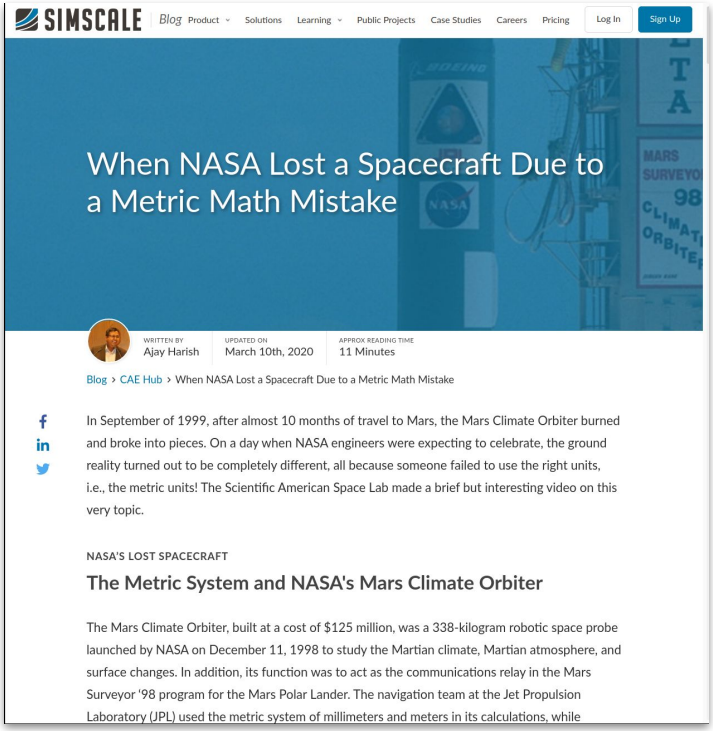
Does this program compile?

```
void processRequest() {
  @Tainted String input = getUserInput();
  if (input.equals("little bobby drop tables")) {
    input = validate(input);
  }
  executeQuery(input);
}
```

Does this program compile? No.

```
void processRequest() {
  @Tainted String input = getUserInput();
  if (input.equals("little bobby drop tables")) {
    input = validate(input); // @Untainted
  }
  executeQuery(input); // @Tainted
}
```

Remember the Mars Climate Orbiter incident from 1999?



NASA's Mars Climate Orbiter (cost of $327 million) was lost because of a discrepancy between use of metric unit Newtons and imperial measure Pound-force.

https://www.simscale.com/blog/2017/12/nasa-mars-climate-orbiter-metric/

Units Checker identifies physical unit inconsistencies

- Guarantees that operations are performed on the same kinds and units
- Kind annotations
  - `@Acceleration, @Angle, @Area, @Current, @Length, @Luminance, @Mass, @Speed, @Substance, @Temperature, @Time`
- SI unit annotation
  - `@m, @km, @mm, @kg, @mPERs, @mPERs2, @radians, @degrees, @A, ...`

https://www.nist.gov/pml/weights-and-measures/metric-si/si-units

```java
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

void demo() {
  @m int x;
  x = 5 * m;

  @m int meters = 5 * m;
  @s int seconds = 2 * s;

  @mPERs int speed = meters / seconds;
  @m int foo = meters + seconds;
  @s int bar = seconds - meters;
}
```

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

void demo() {
  @m int x;
  x = 5 * m;

  @m int meters = 5 * m;
  @s int seconds = 2 * s;

  @mPERs int speed = meters / seconds;
  @m int foo = meters + seconds;
  @s int bar = seconds - meters;
}
```

`@m` indicates that `x` represents meters

To assign a unit, multiply appropriate unit constant from `UnitTools`

Does this program compile?

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

void demo() {
  @m int x;
  x = 5 * m;

  @m int meters = 5 * m;
  @s int seconds = 2 * s;

  @mPERs int speed = meters / seconds;
  @m int foo = meters + seconds;
  @s int bar = seconds - meters;
}
```

`@m` indicates that `x` represents meters

To assign a unit, multiply appropriate unit constant from `UnitTools`

Does this program compile?

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

void demo() {
  @m int x;
  x = 5 * m;

  @m int meters = 5 * m;
  @s int seconds = 2 * s;

  @mPERs int speed = meters / seconds;
  @m int foo = meters + seconds;
  @s int bar = seconds - meters;
}
```

`@m` indicates that `x` represents meters

To assign a unit, multiply appropriate unit constant from `UnitTools`

Does this program compile? No.

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

void demo() {
  @m int x;
  x = 5 * m;

  @m int meters = 5 * m;
  @s int seconds = 2 * s;

  @mPERs int speed = meters / seconds;
  @m int foo = meters + seconds;
  @s int bar = seconds - meters;
}
```

Addition and subtraction between meters and seconds is physically meaningless

Checker Framework: Limitations

- **Can only analyze code that is annotated**
  - Requires that dependent libraries are also annotated
  - Can be tricky, but not impossible, to retrofit annotations into existing codebases
- Only considers the signature and annotations of methods
  - Doesn't look at the implementation of methods that are being called
- Beware of dynamically generated code!
  - Spring Framework
- Can produce false positives!
  - Byproduct of necessary approximations

What static analysis tools should I use?

The best QA strategies employ a combination of tools

How Many of All Bugs Do We Find? A Study of Static Bug Detectors

Andrew Habib
andrew.a.habib@gmail.com
Department of Computer Science
TU Darmstadt
Germany

Michael Pradel
michael@binaervarianz.de
Department of Computer Science
TU Darmstadt
Germany

ABSTRACT

Static bug detectors are becoming increasingly popular and are widely used by professional software developers. While most work on bug detectors focuses on whether they find bugs at all, and on how many false positives they report in addition to legitimate warnings, the inverse question is often neglected: How many of all real-world bugs do static bug detectors find? This paper addresses this question by studying the results of applying three widely used static bug detectors to an extended version of the Defects4J dataset that consists of 15 Java projects with 594 known bugs. To decide which of these bugs the tools detect, we use a novel methodology that combines an automatic analysis of warnings and bugs with a manual validation of each candidate of a detected bug. The results of the study show that: (i) static bug detectors find a non-negligible amount of all bugs, (ii) different tools are mostly complementary to each other, and (iii) current bug detectors miss the large majority of the studied bugs. A detailed analysis of bugs missed by the static detectors shows that some bugs could have been found by variants of the existing detectors, while others are domain-specific problems that do not match any existing bug pattern. These findings help potential users of such tools to assess their utility, motivate and outline directions for future work on static bug detection, and provide a basis for future comparisons of static bug detection with other bug finding techniques, such as manual and automated testing.

International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238213

1 INTRODUCTION

Finding software bugs is an important but difficult task. For average industry code, the number of bugs per 1,000 lines of code has been estimated to range between 0.5 and 25 [21]. Even after years of deployment, software still contains unnoticed bugs. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [8, 24]. Unfortunately, a single bug can cause serious harm, even if it has been subsisting for a long time without doing so, as evidenced by examples of software bugs that have caused huge economic loses and even killed people [17, 28, 46].

Given the importance of finding software bugs, developers rely on several approaches to reveal programming mistakes. One approach is to identify bugs during the development process, e.g., through pair programming or code review. Another direction is testing, ranging from purely manual testing over semi-automated testing, e.g., via manually written but automatically executed unit tests, to fully automated testing, e.g., with UI-level testing tools. Once the software is deployed, runtime monitoring can reveal so far missed bugs, e.g., collect information about abnormal runtime

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
Total:	**31**
Total of 27 unique bugs	

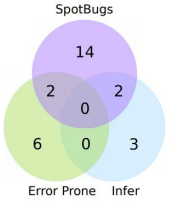


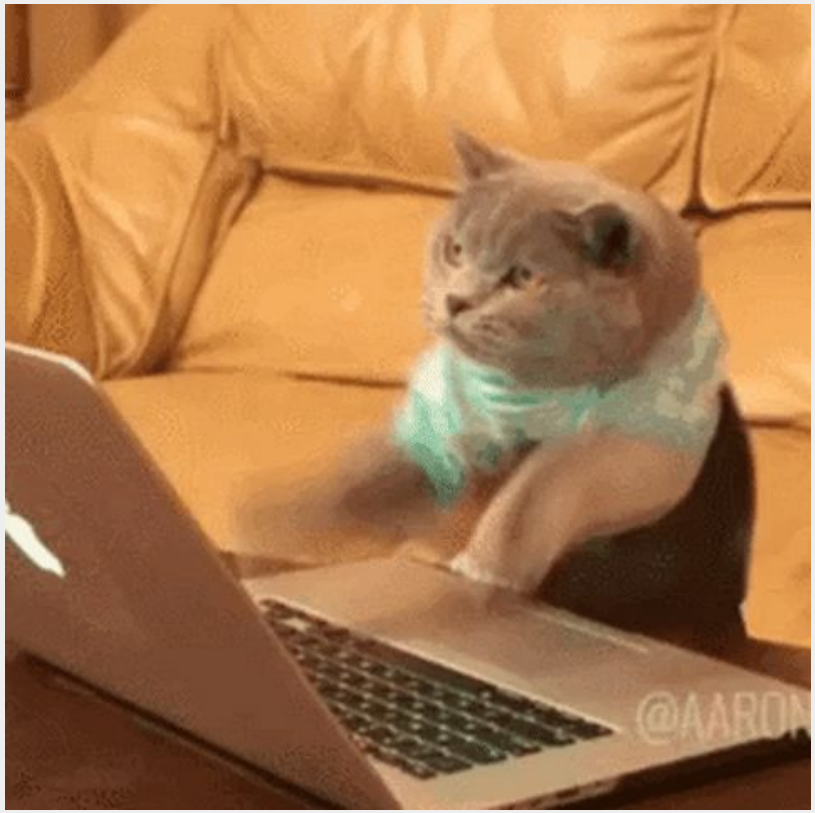
Figure 4: Total number of bugs found by all three static checkers and their overlap.

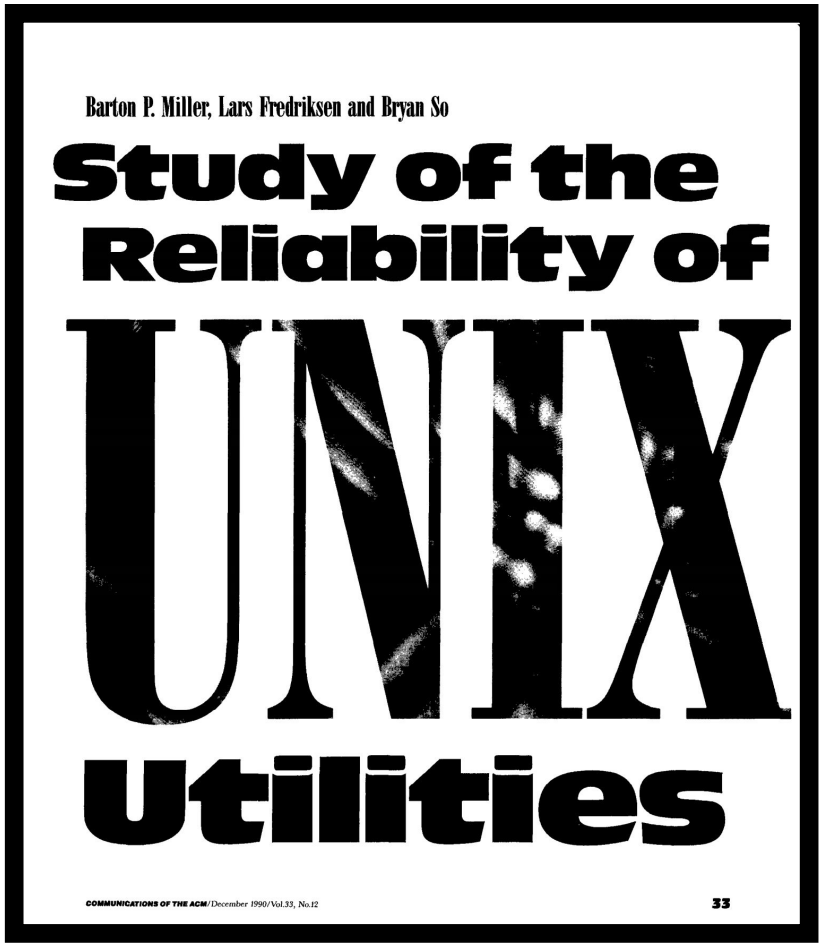
https://software-lab.org/publications/ase2018_static_bug_detectors_study.pdf

Dynamic analysis

Fuzz Testing

Security and Robustness

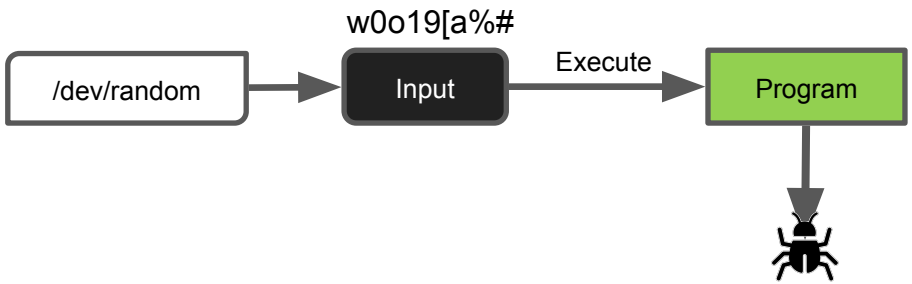




Communications of the ACM (1990)

> On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash.

Fuzz Testing



A 1990 study found crashes in:
adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi

Common Fuzzer-Found Bugs in C/C++

Causes: incorrect arg validation, incorrect type casting, executing untrusted code, etc.

Effects: buffer-overflows, memory leak, division-by-zero, use-after-free, assertion violation, etc. (“crash”)

Impact: security, reliability, performance, correctness

Strengths and Limitations

- **Exercise**: Write down two strengths and two weaknesses of fuzzing.
  Bonus: Write down one or more assumptions that fuzzing depends on.

Strengths and Limitations

Strengths:

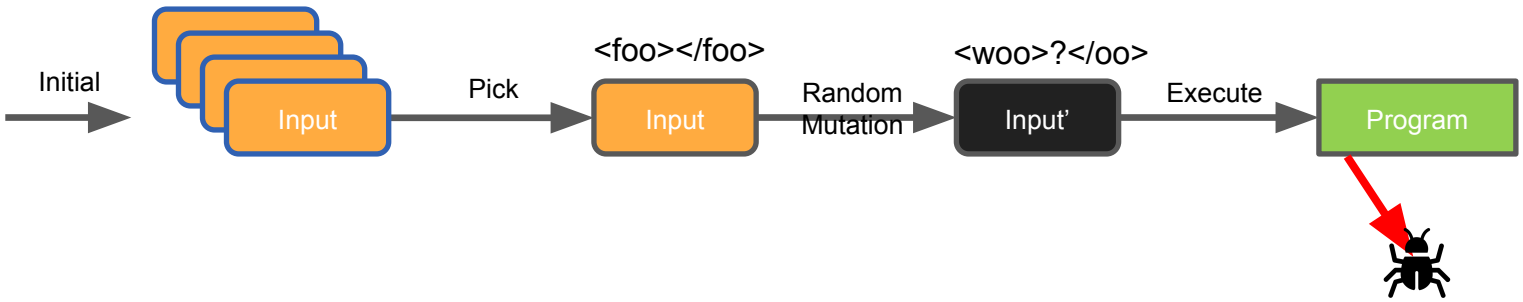
- Cheap to generate inputs
- Easy to debug when a failure is identified

Limitations:

- Randomly generated inputs don’t make sense most of the time.
  - E.g. Imagine testing a browser and providing some ”input” HTML randomly:
- Unlikely to exercise interesting behavior in the web browser
- Can take a long time to find bugs. Not sure when to stop.

S3D

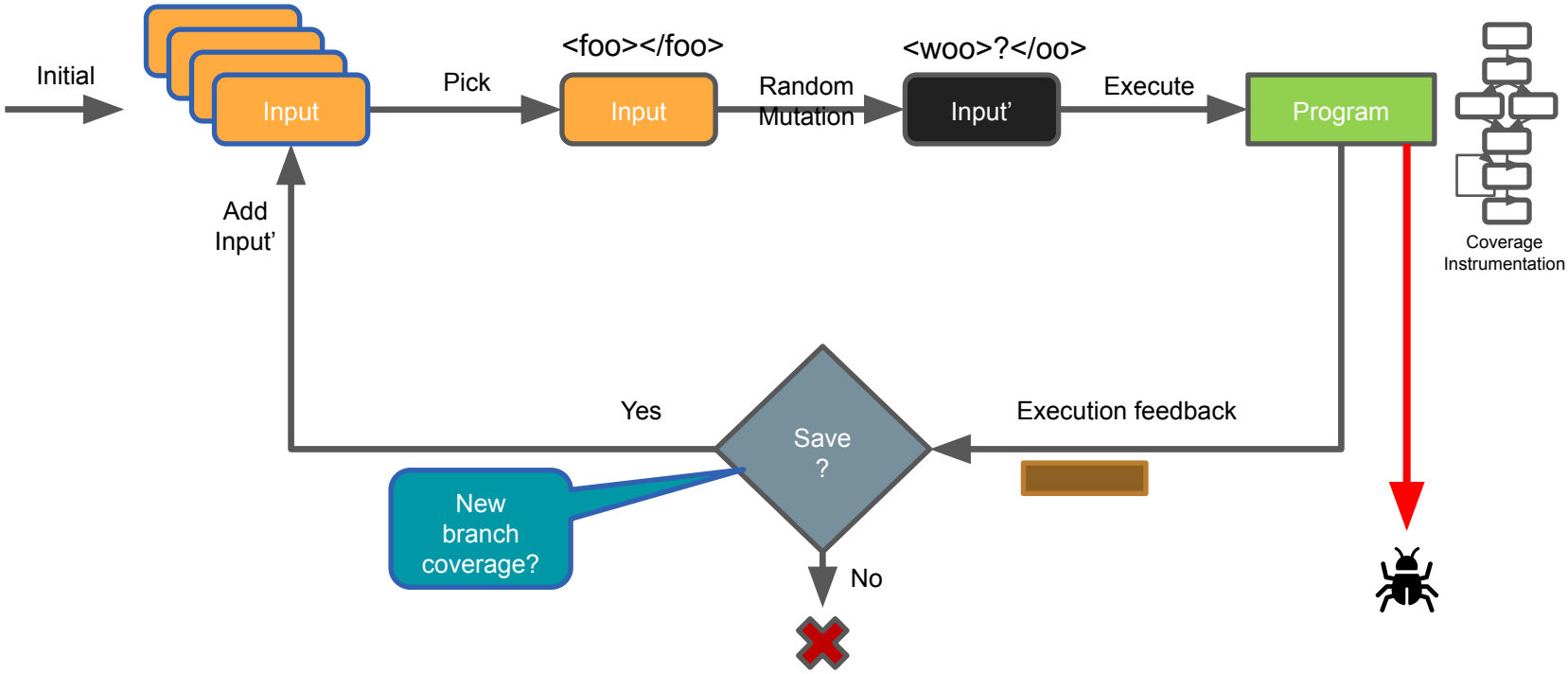
Mutation-Based Fuzzing (e.g. Radamsa)



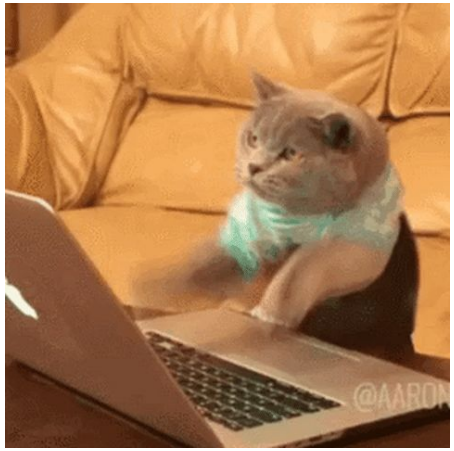
Mutation Heuristics

- Binary input
  - Bit flips, byte flips
  - Change random bytes
  - Insert random byte chunks
  - Delete random byte chunks
  - Set randomly chosen byte chunks to *interesting* values e.g. INT_MAX, INT_MIN, 0, 1, -1, …
- Text input
  - Insert random symbols relevant to format (e.g. “<“ and “>” for xml)
  - Insert keywords from a dictionary (e.g. “<project>” for Maven POM.xml)
- GUI input
  - Change targets of clicks
  - Change type of clicks
  - Select different buttons
  - Change text to be entered in forms
  - … Much harder to design

Coverage-Guided Fuzzing (e.g. AFL)



Now that you can do better than this:



How do you make programs *“crash gracefully”* when a bug is encountered?

Automatic Oracles: Sanitizers

- Address Sanitizer (ASAN) ***
- LeakSanitizer (comes with ASAN)
- Thread Sanitizer (TSAN)
- Undefined-behavior Sanitizer (UBSAN)

https://github.com/google/sanitizers

AddressSanitizer

```
int get_element(int* a, int i) {
   return a[i];
}
```

Is it null?

```
int get_element(int* a, int i) {
   if (a == NULL) abort();
   return a[i];
}
```

Is the access out of bounds?

```
int get_element(int* a, int i) {
   if (a == NULL) abort();
   region = get_allocation(a);
   if (in_heap(region)) {
     low, high = get_bounds(region);
     if ((a + i) < low || (a +i) > high) {
       abort();
     }
   }
   return a[i];
}
```

Is this a reference to a stack-allocated variable after return?

```
int get_element(int* a, int i) {
   if (a == NULL) abort();
   region = get_allocation(a);
   if (in_stack(region)) {
     if (popped(region)) abort();
     …
   }
   if (in_heap(region)) { ... }
   return a[i];
}
```

S3D

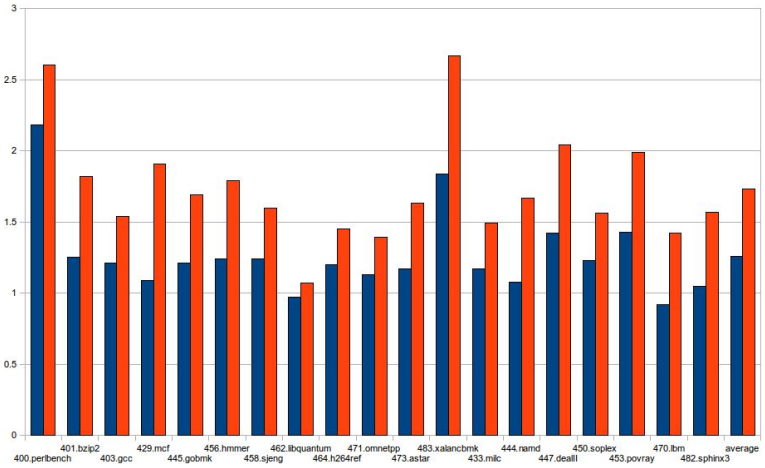
AddressSanitizer

https://github.com/google/sanitizers/wiki/AddressSanitizer

Asan is a memory error detector for C/C++. It finds:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Slowdown about 2x on SPEC CPU 2006



S3D

Testing Performance

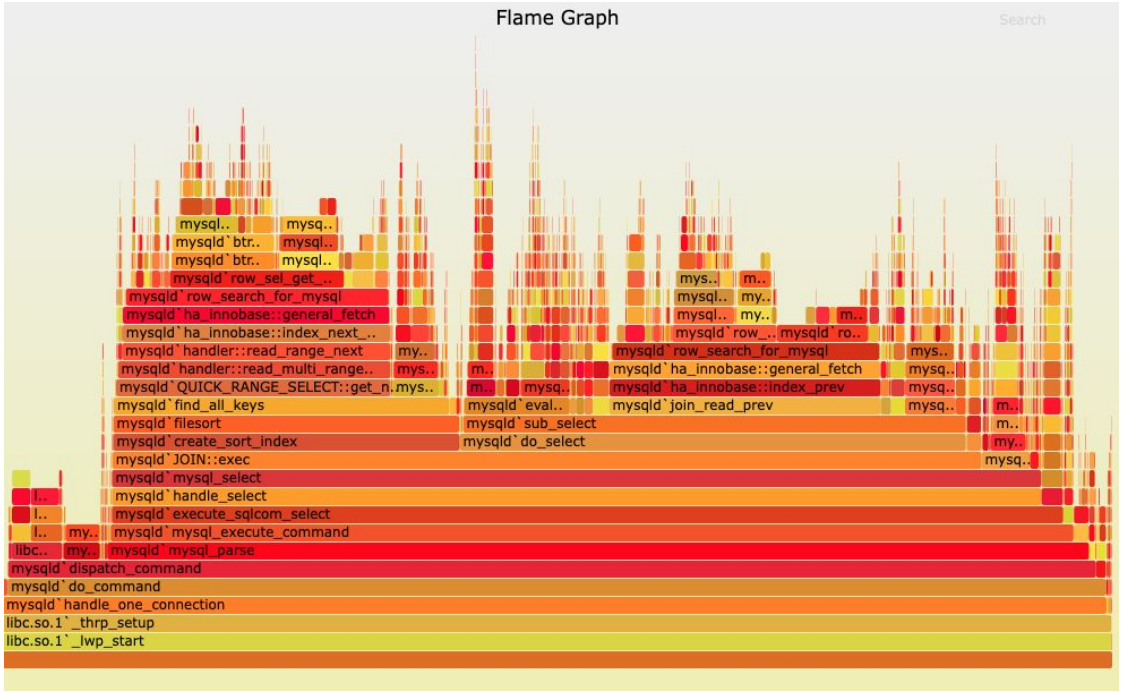
Performance Testing

- Goal: Identify *performance bugs*. What are these?
  - Unexpected bad performance on some subset of inputs
  - Performance degradation over time
  - Difference in performance across versions or platforms

- Not as easy as functional testing. What's the oracle?
  - Fast = good, slow = bad // but what's the threshold?
  - How to get reliable measurements?
  - How to debug where the issue lies?

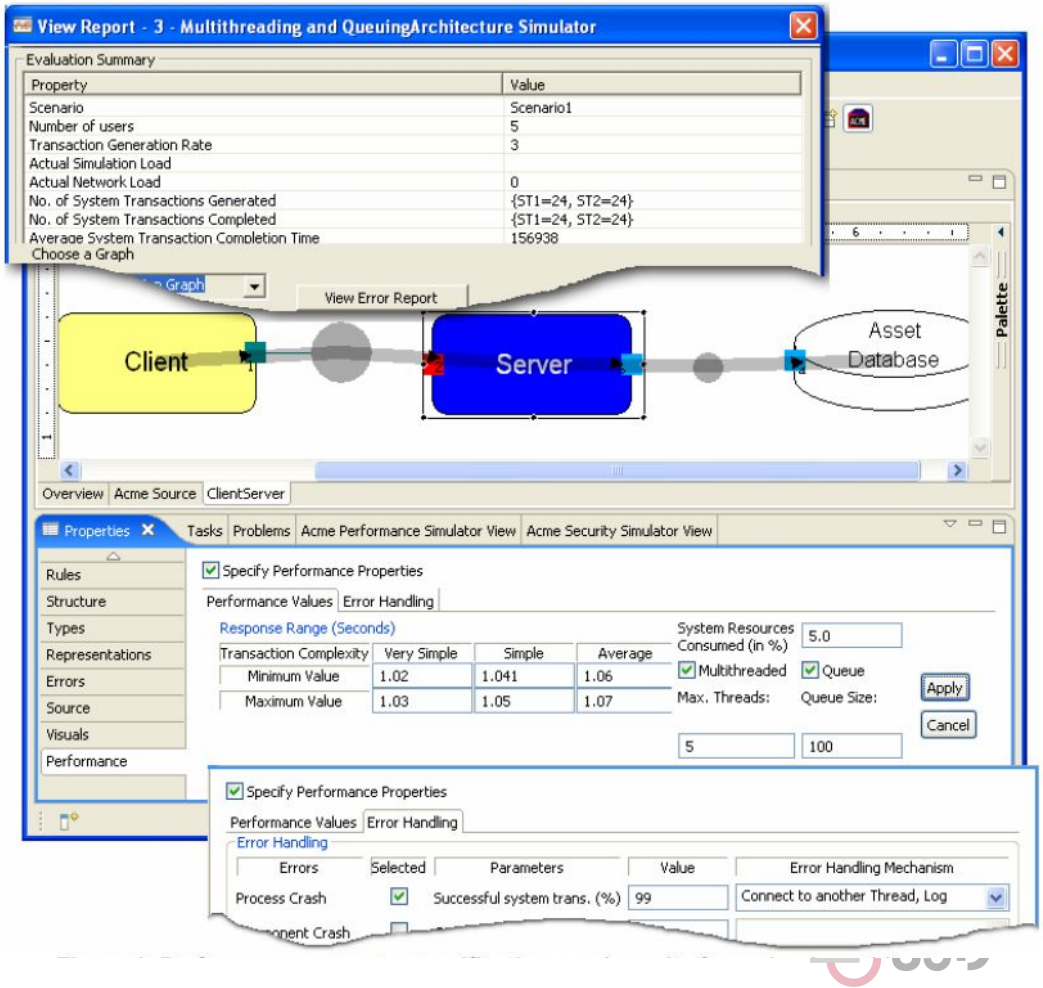
Profiling

- Finding bottlenecks in execution time and memory
- Flame graphs are a popular visualization of resource consumption by call stack.



Performance-driven Design

- Modeling and simulation
  - e.g. queuing theory
- Specify load distributions and derive or test configurations



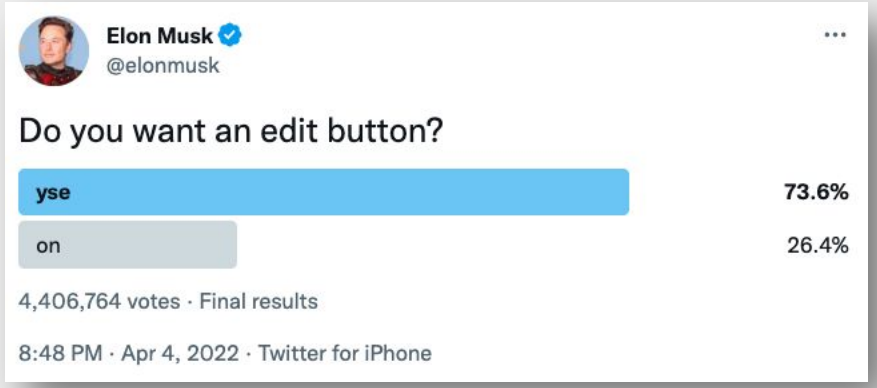
Stress testing

- Robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.

Soak testing

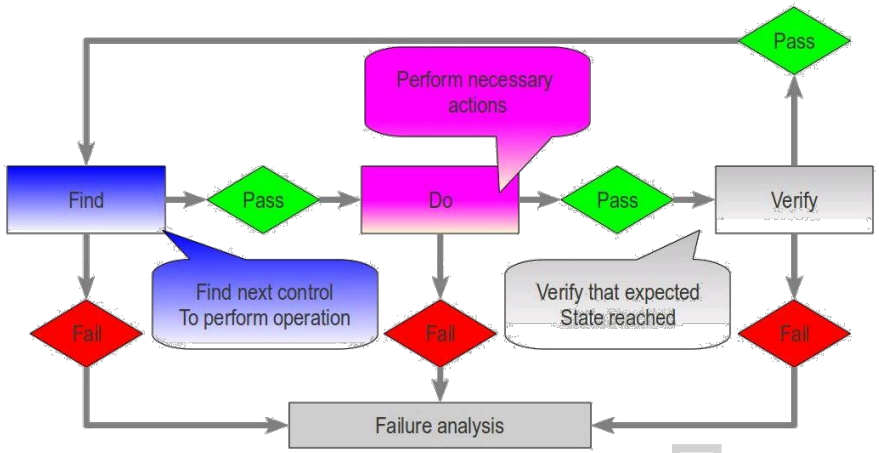
- **Problem**: A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure (also motivates static/dynamic analysis, but we'll talk about that later).
- Soak testing: testing a system with a significant load over a significant period of time (*positive*).
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.

Testing Usability



Automating GUI/Web Testing

- This is hard
- Capture and Replay Strategy
  - mouse actions
  - system events
- Test Scripts: (click on button labeled "Start" expect value X in field Y)
- Lots of tools and frameworks
  - e.g. Selenium for browsers



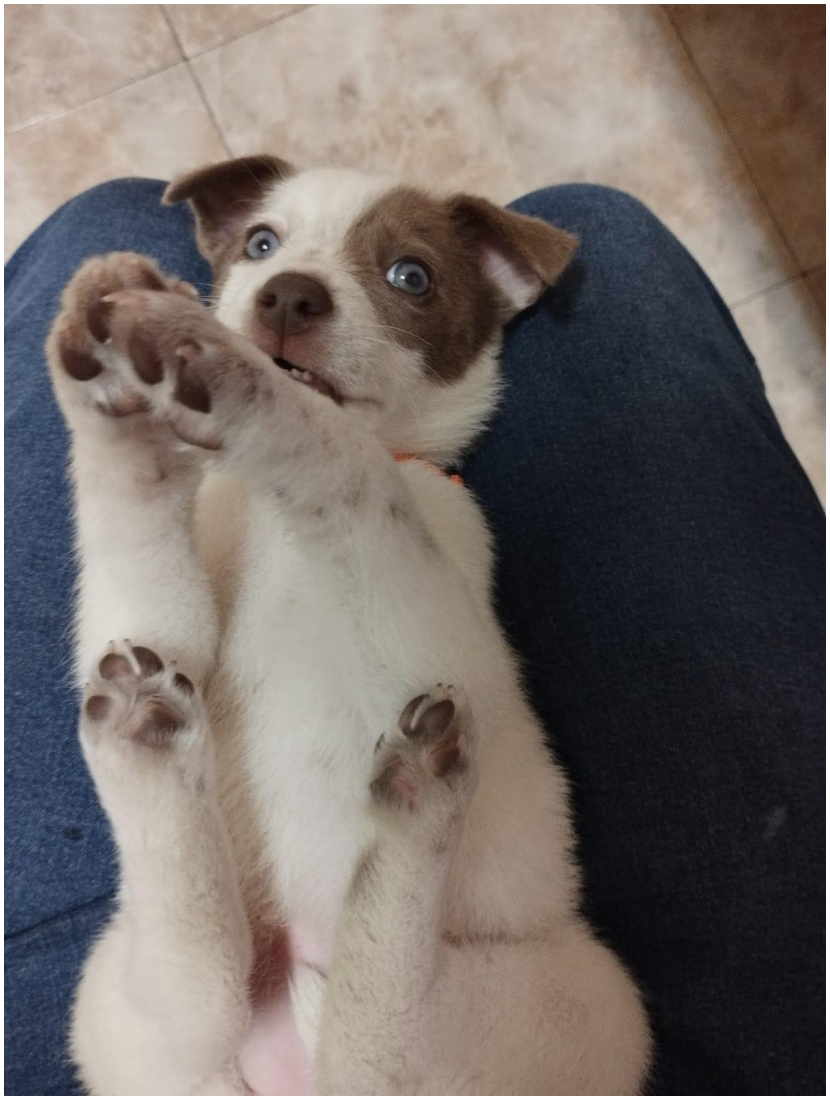
Usability: A/B testing

- Controlled randomized experiment with two variants, A and B, which are the control and treatment.
- One group of users given A (current system); another random group presented with B; outcomes compared.
- Often used in web or GUI-based applications, especially to test advertising or GUI element placement or design decisions.

Example

- A company sends an advertising email to its customer database, varying the photograph used in the ad...

Example: group A (99% of users)



Act now!

Sale ends soon!

Example: group B (1%)

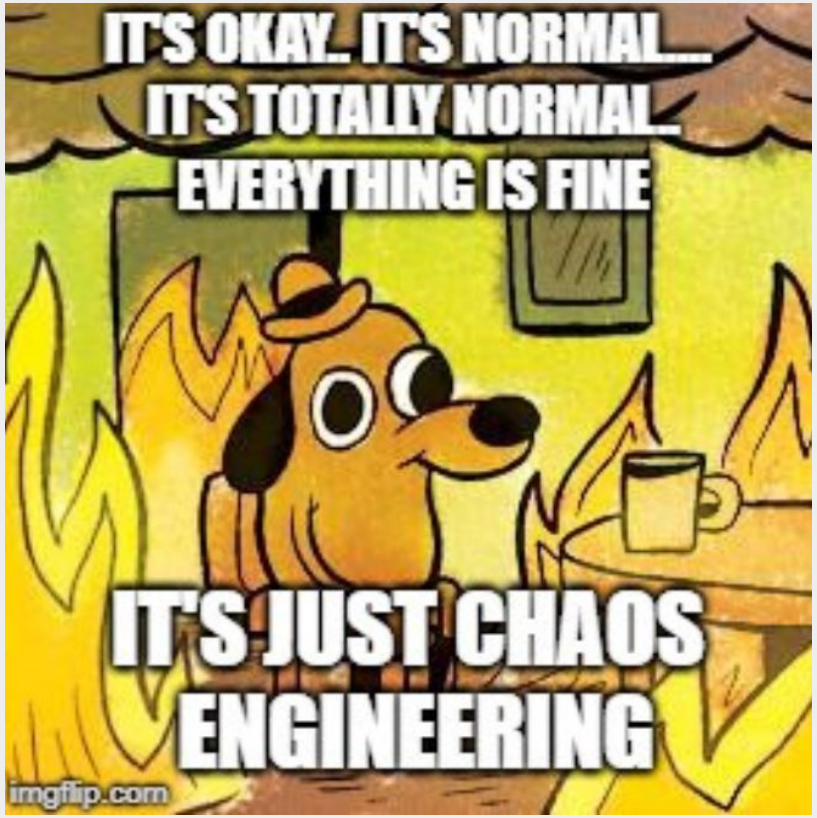
Act now!
Sale ends soon!

A/B Testing

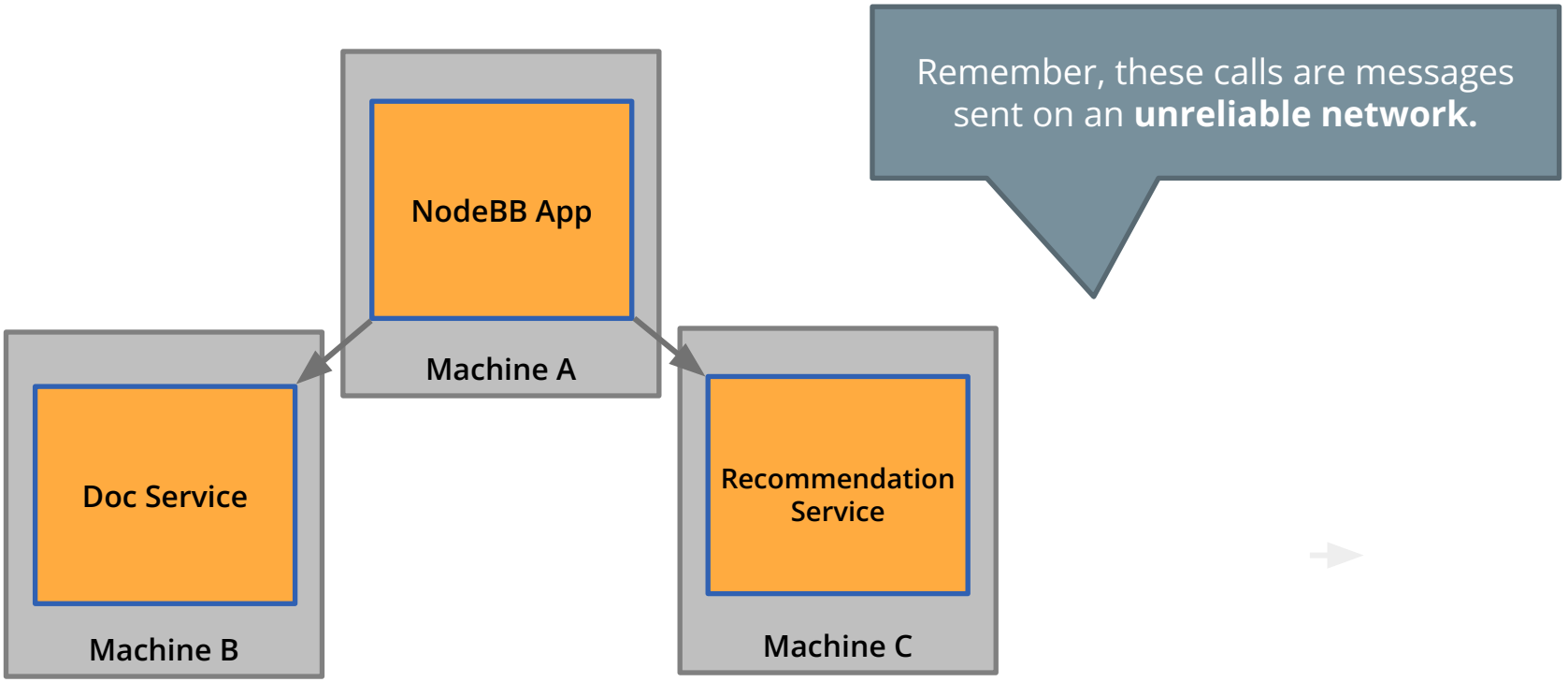
- Requires good metrics and statistical tools to identify significant differences.
- E.g. clicks, purchases, video plays
- Must control for confounding factors

Microservice Failures and Chaos Engineering

Slides credit: Christopher Meiklejohn



Microservice Application



Failures in Microservice Architectures

1. Network may be partitioned
2. Server instance may be down
3. Communication between services may be delayed
4. Server could be overloaded and responses delayed
5. Server could run out of memory or CPU

Where Do We Start?

How do we even **begin to test these scenarios?**

Are there any **techniques** or **software** that can be used to test these types of failures?

Chaos Engineering!

S3D

What is chaos engineering?

- "Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.“

  principlesofchaos.org

S3D

Why would you break things on purpose?

S3D

Game Days

Purposely **injecting failures** into critical systems in order to:

- Identify **flaws** and “latent defects”
- Identify **subtle dependencies** (which may or may not lead to a flaw/defect)
- Prepare a **response** for a disastrous event

Comes from “resilience engineering” typical in high-risk industries

Practiced by Amazon, Google, Microsoft, Etsy, Facebook, Flickr, etc.

S3D

Game Days

Large-scale applications are built on and with **“unreliable”** components

Failure is inevitable (fraction of percent; at Google scale, ~multiple times)

Goals:

- **Preemptively trigger** the failure, observe, and fix the error
- Script testing of **previous failures** and ensure system remains resilient
- Build the necessary relationships between teams **before** disaster strikes

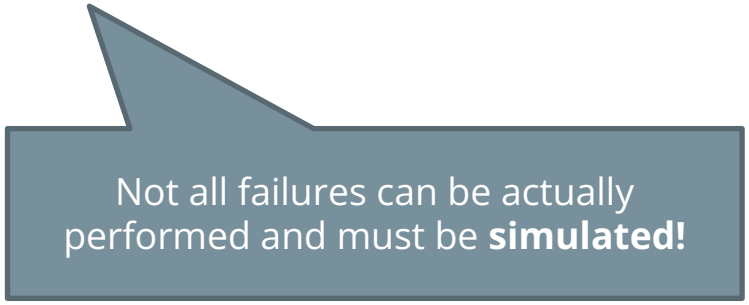
S3D

Example: Amazon GameDay

Full data center destruction (Amazon EC2 region)

- No advanced notice of **which** data center will be taken offline
- No notice of **when** the data center will be taken offline
- Only advance notice (months) that a GameDay **will be happening**
- **Real failures in the production environment**

Discovered **latent defect** where the monitoring infrastructure responsible for detecting errors and paging employees **was located in the zone of the failure!**



S3D

Other examples: Google

Terminate network in Sao Paulo for testing:

- Hidden dependency takes down links in Mexico which would have remained undiscovered without testing

Turn off data center to find that machines won't come back:

- Ran out of DHCP leases (for IP address allocation) when a large number of machines come back online unexpectedly.

Netflix is another heavy cloud user…

Significant deployment in Amazon Web Services in order to remain **elastic** in times of high and low load (first public, 100% w/o content delivery.)

Pushes code into production and modifies runtime configuration hundreds of times a day

Key metric: **availability**

SPS is the primary indicator of the system's overall health.

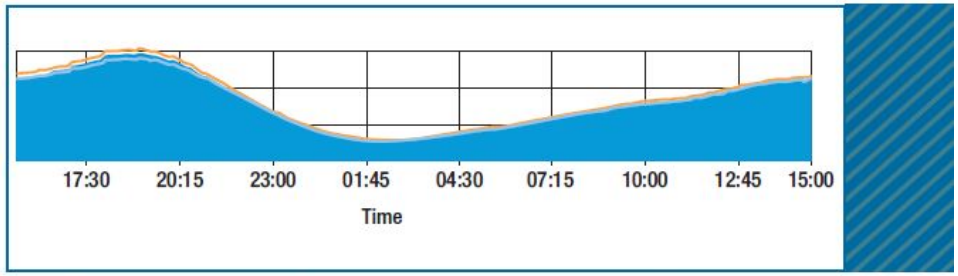
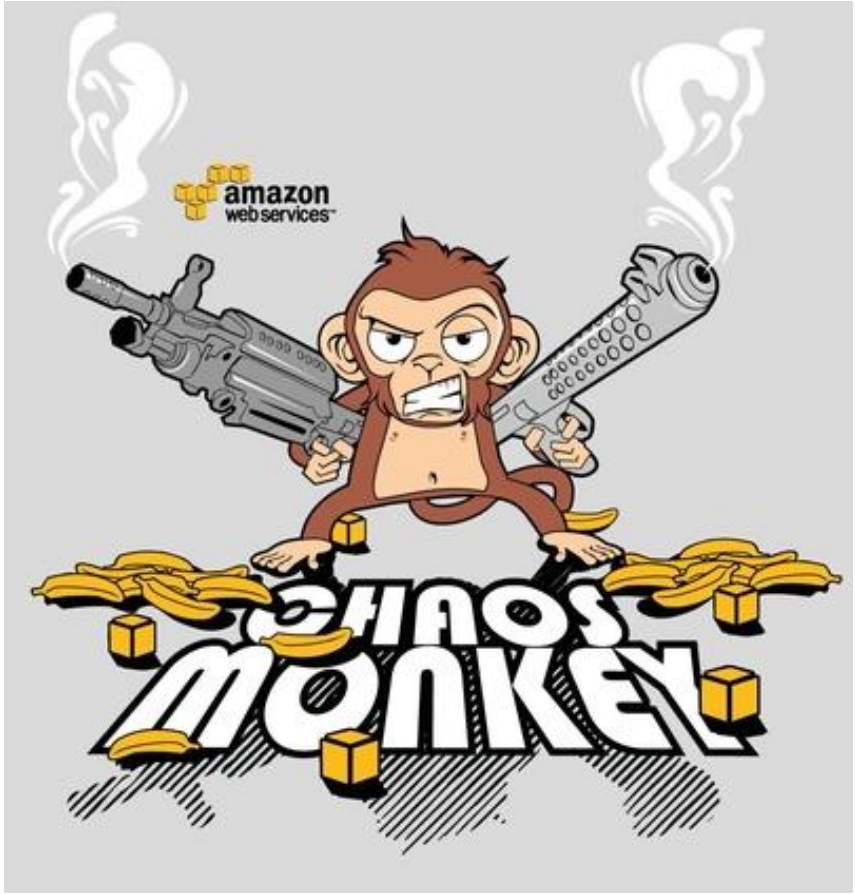


FIGURE 2. A graph of SPS ([stream] starts per second) over a 24-hour period. This metric varies slowly and predictably throughout a day. The orange line shows the trend for the prior week. The *y*-axis isn't labeled because the data is proprietary.

Chaos monkey/Simian army

- A Netflix infrastructure testing system.
- “Malicious” programs randomly trample on components, network, data-centers, AWS instances...
  - Force failure of components to make sure that the system architecture is resilient to unplanned/random outages.
- Netflix has open-sourced their chaos monkey code.



Exit Ticket

- What is the primary objective of fuzz testing in software development?
- primary goal of chaos engineering in the context of software systems?