Fuzz Testing and Input Generation

17-313 Fall 2026

Foundations of Software Engineering

https://cmu-17313q.github.io

Eduardo Feo Flushing

Administrivia

- Team survey due tomorrow (Monday). Don’t forget.
- Slides due tonight. Presentation tomorrow is 10 minutes: plan for less, be concise.

Learning Goals

- Explain why exhaustive testing is infeasible and why random input generation is a practical alternative.
- Compare mutation-based and coverage-guided fuzzing strategies.
- Identify the oracle used by a testing or fuzzing approach and judge whether it is strong or weak.
- Write appropriate preconditions, postconditions, and invariants as executable assertions and use them as testing oracles.
- Explain how fuzzing combines input generation with an oracle, and how stronger oracles can reveal bugs beyond crashes.

What are the challenges and limitations of traditional, example-based testing?

Principles of Testing

A quick recap before we get to fuzzing

Principles of Testing #1: Testing Shows the Presence, Not the Absence, of Defects

- Testing shows the presence of defects.
- Testing does not show the absence of defects!
- *"No test team can achieve 100% defect detection effectiveness."*

Edsger W. Dijkstra

Dijkstra on Bugs

NOTES

This is the most common misconception about testing: passing tests is evidence of correctness for the cases you tried, not proof that no bugs remain. Fuzzing makes this especially visible: it can run millions of inputs without finding a crash, and that still proves nothing about the inputs it didn't try.

What About Exhaustive Testing?

Idea: try all values!

- `age` : `int` (2 to 117) years
- `ride_datetime` : `datetime` (hh:mm + M/D/Y)
- `ride_duration` : `int` (in minutes, 1 to 2 hours)
- `is_public_holiday` : `bool` (2 values)

116 × 1440 (minutes per day) × 1826 (days in the next 5 years) × 120 (ride duration) × 2

~ 72 billion test cases

NOTES

Recall this example from the testing lecture: even after adding a fourth variable, 72 billion is still technically within reach of modern hardware. But please, don't do it. Running (and maintaining) a suite that size buys you almost nothing beyond a much smaller, well-chosen set of cases, and it costs a lot of time.

What About Exhaustive Testing?

Exhaustive testing is usually impractical, even for a trivially small problem.

Key problem: choosing the test suite.

- **Small enough** to finish in a useful amount of time.
- **Large enough** to provide a useful amount of validation.

Alternative: **heuristics**

Principles of Testing #2: Exhaustive Testing Is Impossible

```
def is_valid_email(email: str) -> bool:
    ...
```

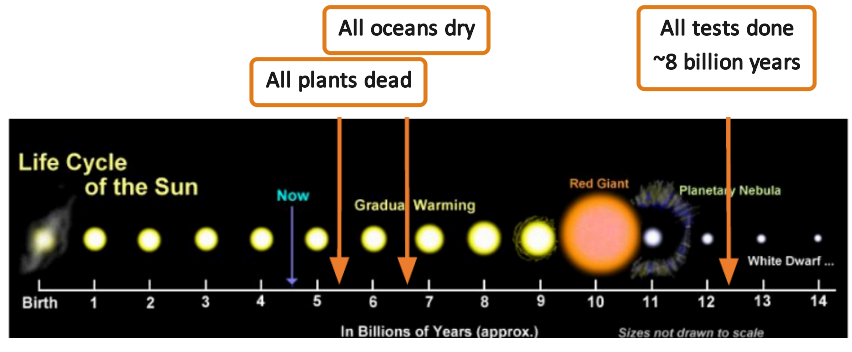
- A simple function, 1 input, string, max.

  26 lowercase characters plus symbols

  `(@, ., _, -)`

- Assume we can use 1 zettaFLOPS: 10^{21}

  tests per second



Principles of Testing #3: Start Testing Early

- To let tests guide design.
- To get feedback as early as possible.
- To find bugs when they are cheapest to fix.
- To find bugs when they have caused the least damage.

Principles of Testing #4: Defects Are Usually Clustered

- "Hot" components requiring frequent change, bad habits, poor developer practices, tricky logic, business uncertainty, innovative code, size, etc.
- Use as a heuristic to focus test effort.

NOTES

This is also why fuzzing tends to be aimed at parsers and input-handling code specifically: code that touches untrusted, highly variable input is exactly the kind of "hot," tricky-logic code where defects cluster.

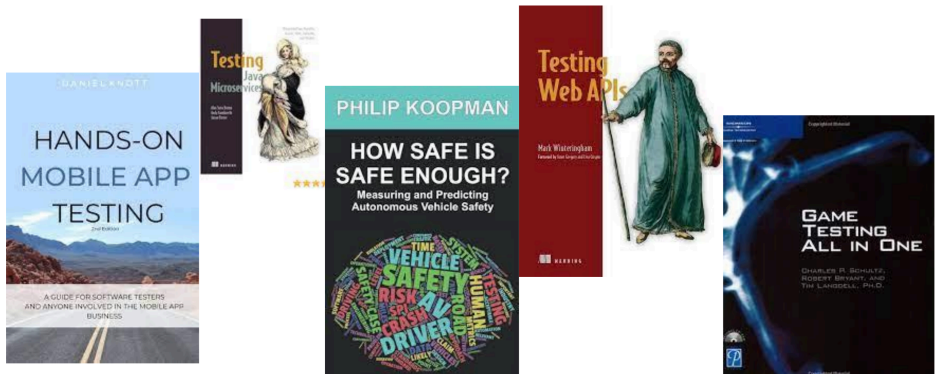
Principles of Testing #5: The Pesticide Paradox

"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."

- Re-running the same test suite again and again on a changing program gives a false sense of security.
- Variation in testing is needed.

Principles of Testing #6: Testing Is Context-Dependent

- A safety-critical system, a mobile game, and a one-off script all warrant very different testing effort and rigor.
- What counts as "enough testing" depends on the cost of failure, not on a fixed checklist.



Principles of Testing #7: Verification Is Not Validation

Verification

- Does the software system meet the requirements **specifications**?
- Are we building the **software right**?

Validation

- Does the software system meet the **user's real needs**?
- Are we building the **right software**?

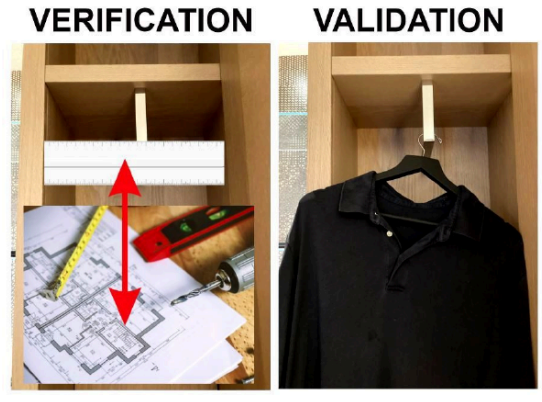
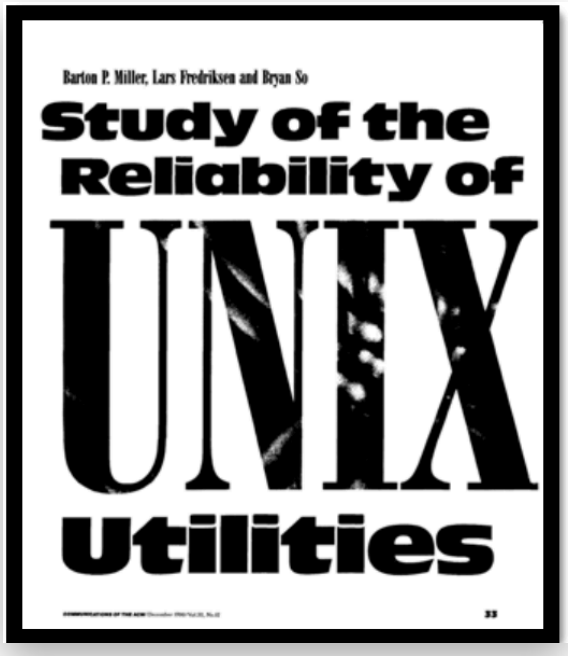


Image credit: Philip Koopman

Disclaimer

Many (unintentional) references to animals ahead

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."

How can we identify these bugs?

NOTES

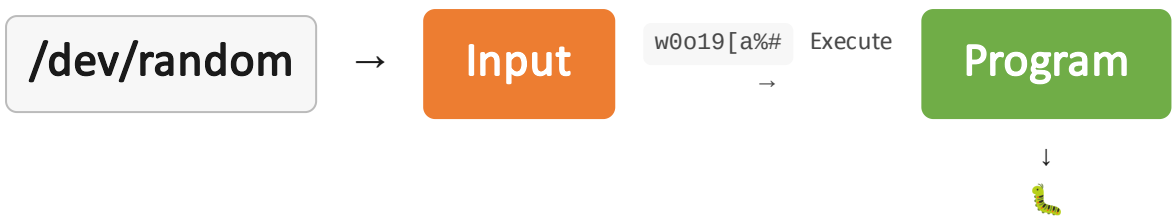
Miller, Fredriksen, and So fed random noise into standard UNIX utilities and crashed a large fraction of them. This is the study that popularized fuzz testing as a technique.

Infinite Monkey Theorem

*"a **monkey** hitting keys **at random** on a typewriter **keyboard** for an **infinite amount of time** will almost surely type any given text, including the complete works of **William Shakespeare**."*

https://en.wikipedia.org/wiki/Infinite_monkey_theorem

Fuzz Testing Randomly Generates Inputs and Checks for Program Crashes



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

Cicada

Fun fact: Cicadas evolved 13- and 17-year cycles. Because these are prime numbers, their emergence is less likely to regularly line up with predator cycles.

NOTES

The key idea: no example-based oracle, no hand-picked test cases. Just generate garbage, run it, and see if the program falls over. Crashing on garbage input is never correct behavior, which is what makes "did it crash?" a usable oracle even with zero knowledge of what the program is supposed to do.

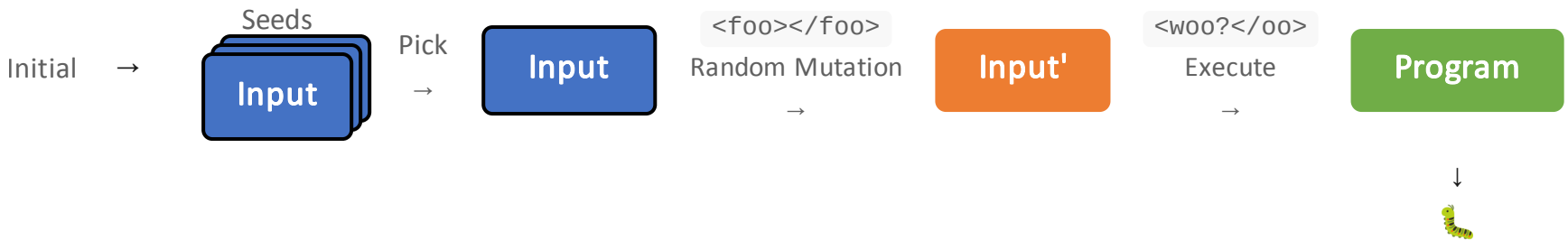
Common Fuzzer-Found Bugs in C/C++

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

Effects: buffer overflows, memory leaks, division by zero, use-after-free, assertion violations, etc. ("crash")

Impact: security, reliability, performance, correctness

Mutation-Based Fuzzing (e.g., Radamsa)



Hercules Beetle

Fun fact: Hercules beetles can lift ~850x their own body weight. That's like me lifting an entire Boeing 737.

https://gitlab.com/akihe/radamsa

Mutation Heuristics

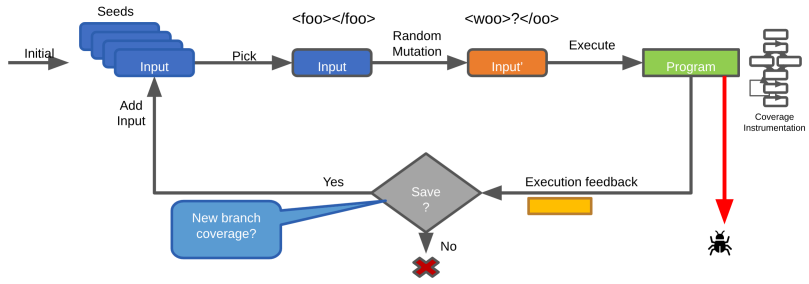
- **Binary input:** bit flips, byte flips; modify, insert, delete random byte chunks; set randomly chosen byte chunks to interesting values, e.g. `INT_MAX`, `INT_MIN`, `0`, `1`, `-1`, …
- **Text input:** insert random symbols relevant to the format (e.g., `<` and `>` for XML); insert keywords from a dictionary (e.g., `<project>` for a Maven `POM.xml`)
- **GUI input:** change click types and targets; change text, click different buttons

```
<html><head><title>Hello</title></head><body>World<br/></body></html>
```

https://www.fuzzingbook.org/html/GreyboxGrammarFuzzer.html

Coverage-Guided Fuzzing (e.g., AFL)

https://lcamtuf.coredump.cx/afl/



Spotted Lanternfly

Fun fact: the spotted lanternfly was first found in the US in Pennsylvania in 2014 and has since spread through Pittsburgh. Squishing them on sight is actively encouraged to slow the invasive species.

NOTES

Mutation-based fuzzing (Radamsa) mutates inputs blindly. Coverage-guided fuzzing (AFL) instruments the program so it can tell whether a mutated input explored new code. Only those inputs that reach new coverage are saved back into the seed pool, so the fuzzer's seed pool evolves toward inputs that exercise more of the program.

Activity: Pick a Scenario

Pick one scenario based on where you are sitting:

- E-commerce web application (front rows)
- Automotive software for self-driving cars (middle rows)
- Mobile gaming application (back rows)

Discuss in groups of 2–3 the applicability of fuzz testing in your scenario, considering:

- One type of input to fuzz.
- One potential vulnerability or bug fuzz testing might uncover.
- One specific challenge in implementing fuzz testing for the scenario.

NOTES

Have each group briefly report back: front rows (e-commerce) tend to focus on input validation and payment/checkout logic; middle rows (self-driving cars) tend to focus on sensor data and safety-critical failure modes; back rows (mobile games) tend to focus on save-state corruption and network play. The point is that the input space, the oracle, and the cost of a missed bug all change with the domain.

Finding Security Bugs = Money

Meta Bug Bounty total rewards to date: **$31,408,827** ($5,401,686 for 2026 alone)

Bug Bounty rewards

All listed amounts are without bonuses. With Hacker Plus, and any applicable bonuses, you can earn up to 30% of the original bounty amount on top of it!

We pay based on maximum security impact found internally, and our highest payouts reflect that.

Total rewards for 2026
$5,401,686

Total rewards to date
$31,408,827

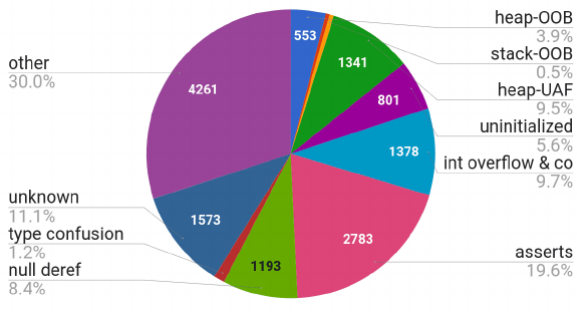
$300K*	$130K*	$30K*	$20K*	$10K*	$5K*	$500*
Mobile RCE WhatsApp Private Processing Muse VM Escape	Account Takeover	Quest Persistent full secure boot bypass	2FA Bypass	Contact point deanonymization	Page admin disclosure	Minimum bounty

Fuzzing in Practice

- After the OpenSSL Heartbleed vulnerability discovered in 2016, Google launched **OSS-Fuzz**.
- Free service for open-source projects: *"The project must have a significant user base and/or be critical to the global IT infrastructure."*
- OSS-Fuzz privately alerts developers to vulnerabilities.

Fuzzing in Practice

- Google uses **ClusterFuzz** to fuzz all of their products; supports multiple fuzzing strategies.
- *As of February 2026, ClusterFuzz has found 30,000+ bugs in Google code (e.g., Chromium).*
- As of May 2025, OSS-Fuzz has helped identify and fix over 13,000 vulnerabilities and 50,000 bugs across 1,000 projects (e.g., nodejs, django, openvpn, openssl).



Fuzz Testing in the AI Era

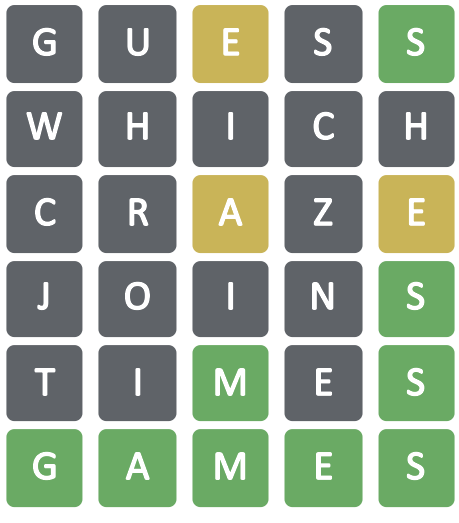
- Fuzz testing is nearly forty years old, yet still not widely adopted outside specialist niches like penetration testing and security.
- Example: Meta's **Sapienz** fuzzes Android apps with genetic-algorithm-based test generation, integrated directly into CI.
- LLMs make it easier to generate the grammars and seed corpora fuzzers need for structured, semantically valid inputs, lowering the barrier to entry.

Richard Gall, "Fuzz-testing in the AI era: rediscovering an old technique for new challenges," Thoughtworks, July 2025.

https://arstechnica.com/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

Activity: Wordle Input Generator

1. How can you generate random inputs for `match_guess`?
2. What inputs are more likely to find bugs in `match_guess`?
3. Describe a strategy for generating these "likely bug" inputs.



NOTES

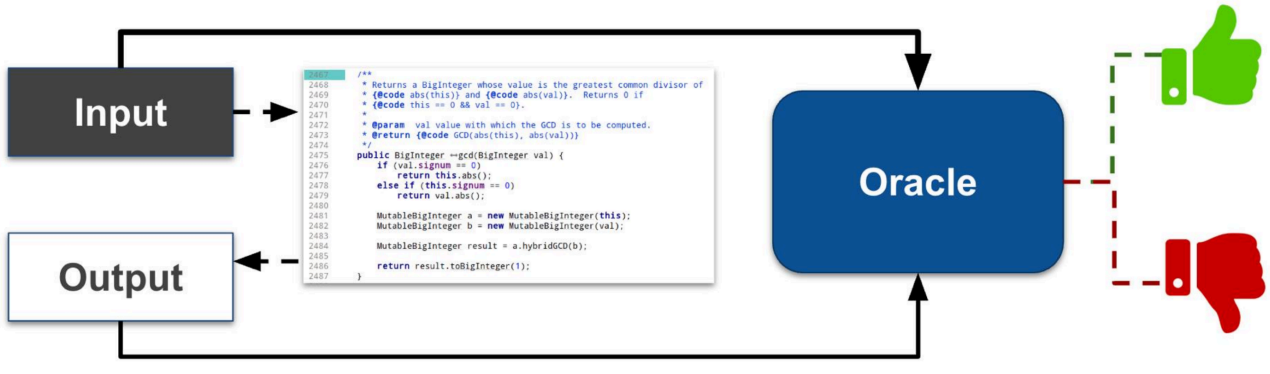
Purely random ASCII strings will almost never match any letter of the secret word, so they mostly exercise the "all black" path. Bugs are more likely at the boundaries: guesses with repeated letters, guesses that share letters with the secret word, wrong-length guesses, non-alphabetic characters, and empty input.

What are the limitations of random input generation?

Oracles

Testing Is Only as Good as Your Oracle

- An oracle decides if behavior is correct for a given input.
  - **Strong oracles** catch bugs that **weak oracles** miss.
  - Designing strong oracles is difficult and often the bottleneck.



Oracle: Assertions in Example-Based Tests

- This is the most common type of oracle in traditional tests.
- These assertions are often hardcoded to a specific test input.
  - Tedious to write for complex outputs (e.g., documents, actions).
  - Can be very brittle (e.g., formatting changes lead to test failures).
  - Non-determinism and environment coupling lead to flaky tests.

```
it('should redirect to login if user is not logged in', async () => {
  const { response, body } = await request.get(`${nconf.get('url')}/me/bookmarks`);
  assert.equal(response.statusCode, 200);
  assert(body.includes('Login to your account'), body.slice(0, 500));
});
```

Oracle: The Program Shouldn’t Crash!

- This is the oracle used by most fuzzing approaches.
- This oracle is a **generic property** that is not tied to any test inputs.
  - That allows us to automatically generate and test any input.
  - But the oracle is weak (i.e., not crashing does not imply correct).
- We can make the oracle slightly stronger by using **sanitizers**.
  - Detects illegal program states that might not cause an immediate crash; instruments the program at compile time (e.g., `-fsanitize=address`).
  - Finds more safety issues but slows down execution / fuzzing, and still doesn’t reveal logic bugs.

Oracle: Assertions in Source Code

- Assertions are **executable specifications**.
  - Document intended behavior (pre/postconditions, invariants).
- This oracle is generic and **not tied to any test inputs**.
  - If we add assertions, we can use fuzzing to find some logic bugs!

```
function toUSD(amountCents: number): string {
  assert(Number.isInteger(amountCents), 'amount must be integer cents');
  assert(amountCents >= 0, 'amount must be non-negative');
  const dollars = (amountCents / 100).toFixed(2);
  return `$${dollars}`;
}
```

https://blog.regehr.org/archives/1091 · https://nullprogram.com/blog/2022/06/26

Assertions Catch Infections Earlier

- Finds more bugs (e.g., during fuzzing) and helps to localize them.

Sane state → **Bug executes** → **Infected state** → **Visible failure**

An assertion placed right after the bug executes catches the infection immediately, instead of waiting for it to propagate into a visible failure much later, far from its actual cause.

https://www.whyprogramsfail.com/pdf/AssertingExpectations.pdf

Assertions Should Always Be True Unless You Have a Bug in Your Code

- Assertions state invariants: conditions that must always hold if the program is correct (e.g., impossible states, internal consistency).
  - **Never rely on asserts for control flow or user-visible behavior.**
  - Make sure that your assertions **don’t contain side effects.**
- Use exceptions and returns for errors that can reasonably happen and should be handled (e.g., invalid inputs, failed API calls).

Assertions in the Wild: Apache Cassandra

Used to enforce an **invariant** that must hold throughout sorting.

```
// start is the first element not already known sorted (lo <= start <= hi)
private static void binarySort(long[] a, int lo, int hi, int start,
                               LongComparator c) {
    if (DEBUG) assert lo <= start && start <= hi;
    if (start == lo)
        start++;
    for ( ; start < hi; start++) {
        long pivot = a[start];
        int left = lo;
        int right = start;
        if (DEBUG) assert left <= right;
```

github.com/apache/cassandra: LongTimSort.java#L227

Assertions in the Wild: SQLite & LLVM

Used to enforce a **precondition** and find bugs at call sites.

```
/*
** Insert a new entry into the cache. If the
** cache is full, expel the least recently
** used entry. Return SQLITE_OK on success.
**
** Cache entries are stored in age order,
** oldest first.
*/
static int jsonCacheInsert(
  sqlite3_context *ctx,
  JsonParse *pParse
){
  JsonCache *p;

  assert( pParse->zJson!=0 );
  assert( pParse->bJsonIsRCStr );
  assert( pParse->delta==0 );
  p = sqlite3_get_auxdata(ctx, JSON_CACHE_ID);
```

```
m_current_value = value;
return true;
}
return false;
}

bool SetDefaultValue(uint64_t value) {
  assert(value >= m_min_value &&
         value <= m_max_value &&
         "disallowed default value");
  m_default_value = value;
  return true;
}
```

github.com/sqlite/sqlite: json.c#L439 · LLDB `OptionValueUInt64.h`

Assertions in the Wild: Firefox

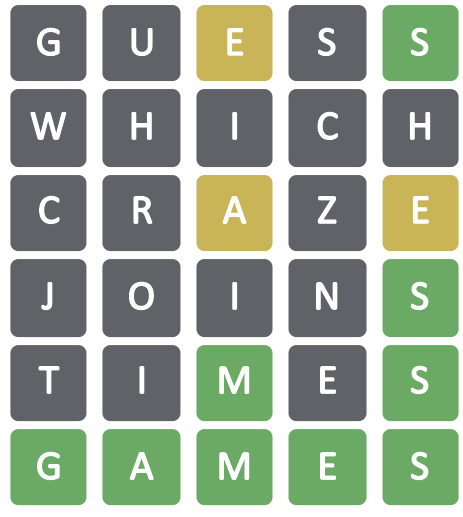
Used to enforce a **postcondition** that makes sure an audio packet is the correct size after processing.

```
namespace mozilla {
void AudioInputProcessing::Process(AudioProcessingTrack* aTrack, ...) {
  // Postconditions of the audio-processing logic.
  MOZ_ASSERT(static_cast<uint32_t>(mSegment.GetDuration()) +
             mPacketizerInput->FramesAvailable() ==
             mPacketizerInput->mPacketSize);
  MOZ_ASSERT(mSegment.GetDuration() >= 1);
  MOZ_ASSERT(mSegment.GetDuration() <= mPacketizerInput->mPacketSize);
}
```

searchfox.org/firefox-main: MediaEngineWebRTCAudio.cpp

Activity: Wordle Oracle

1. What precondition assertions could you add to `mark_guess`?
2. What postcondition assertions could you add?



NOTES

Preconditions: guess and secret are the same length; both are non-empty; both contain only letters. Postconditions: the result has exactly one mark per letter of the guess; the count of green plus yellow marks for any letter never exceeds that letter's count in the secret; a letter absent from the secret is always marked black.