Fuzz and prop-based testing

“Automated” autotests

Agenda

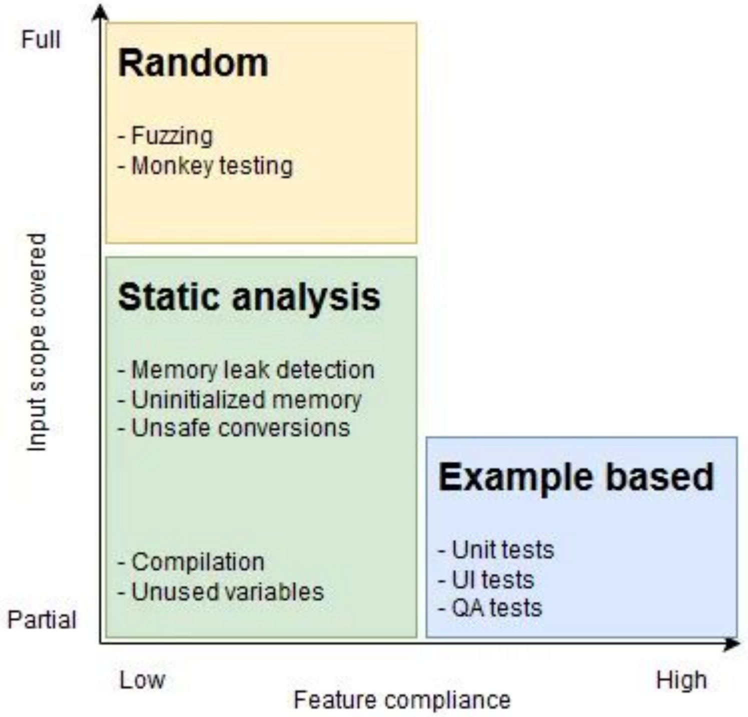
- Fuzzing 101
    - What it is?
- Property based testing
    - Isn’t it the same?

What fuzzing is?

Fuzz testing is an automated software testing technique that involves providing **invalid**, **unexpected**, or **random data** as *inputs* to a computer program.

W

ikipedia



Brief history

- 1950s - punched cards trash-deck technique
- Early 1980s, some mentions of random-testing of computer programs
    - “TheMonkey” generated a random input for classic MacOS apps
- 1988, term *fuzzing* was introduced (link to paper)
- 1991, “crashme”**:** generate random bytes and jump to them
- 2012, ClusterFuzz
- 2014-2015, AFL, libFuzzer, go-fuzz
- 2016, OSS-Fuzz (Google), Project Springfield (Microsoft)

What can be fuzzed?

TLDR; Everything which consumes some sort of an input.

- Network protocols
- Parsers
- Compression
- Media codecs
- Crypto
- Text processing
- Compilers, interpreters, databases, kernels, you name it

Must have for everything which is somewhat **security sensitive** and inputs are crossing trust boundaries.

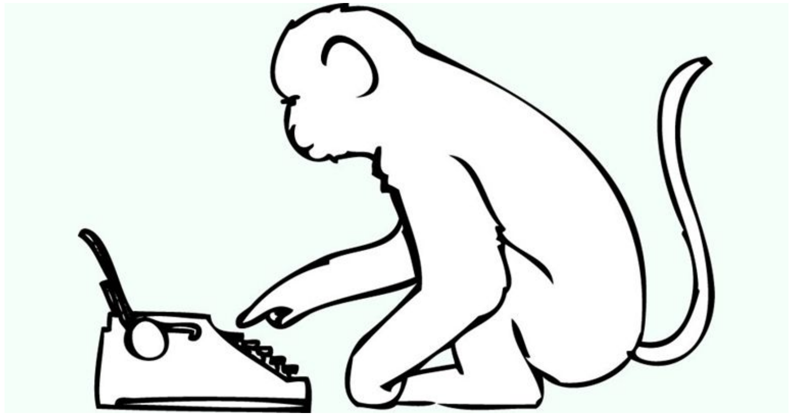
Types of fuzzers

- Data generation approaches
  - Generation based
  - Mutation based
- Aware of input structure or not
- Aware of program structure or not
- In-process or standalone

Data generation approaches

Generation based

- Random (not so effective)
- Grammar based
- Filtering according some rules applied to generated data



Mutation based

- Requires predefined “corpus”
- Derives new inputs by mutating known inputs, and adding *interesting* examples to the ‘pool’ of known inputs
- Initial examples might be mutated to shrink the size

Side note: Sanitizers

Usually applicable to compiled languages.

Injection of additional assertions into program during build time.

Makes fuzzer **way more** sensitive.

Examples:

- AddressSanitizer
- LeakSanitizer
- ThreadSanitizer
- MemorySanitizer

What can be found?

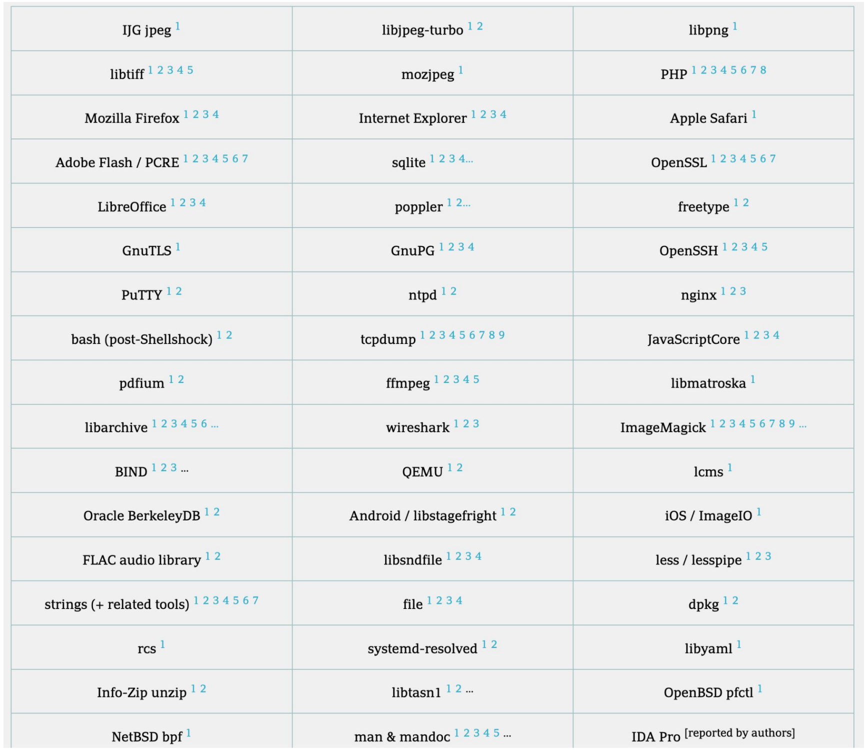
Oh, a lot. :)

Depends on a type of fuzzer list might vary but obvious things are:

- memory leaks
- out-of-bound access
- deadlocks
- nil derefs

Trophies gained by AFL fuzzer (not a full table) ->

“Shellshock” was found by AFL as well.



IJG jpeg [1]	libjpeg-turbo [1 2]	libpng [1]
libtiff [1 2 3 4 5]	mozjpeg [1]	PHP [1 2 3 4 5 6 7 8]
Mozilla Firefox [1 2 3 4]	Internet Explorer [1 2 3 4]	Apple Safari [1]
Adobe Flash / PCRE [1 2 3 4 5 6 7]	sqlite [1 2 3 4...]	OpenSSL [1 2 3 4 5 6 7]
LibreOffice [1 2 3 4]	poppler [1 2...]	freetype [1 2]
GnuTLS [1]	GnuPG [1 2 3 4]	OpenSSH [1 2 3 4 5]
PuTTY [1 2]	ntpd [1 2]	nginx [1 2 3]
bash (post-Shellshock) [1 2]	tcpdump [1 2 3 4 5 6 7 8 9]	JavaScriptCore [1 2 3 4]
pdfium [1 2]	ffmpeg [1 2 3 4 5]	libmatroska [1]
libarchive [1 2 3 4 5 6 ...]	wireshark [1 2 3]	ImageMagick [1 2 3 4 5 6 7 8 9 ...]
BIND [1 2 3 ...]	QEMU [1 2]	lcms [1]
Oracle BerkeleyDB [1 2]	Android / libstagefright [1 2]	iOS / ImageIO [1]
FLAC audio library [1 2]	libsndfile [1 2 3 4]	less / lesspipe [1 2 3]
strings (+ related tools) [1 2 3 4 5 6 7]	file [1 2 3 4]	dpkg [1 2]
rcs [1]	systemd-resolved [1 2]	libyaml [1]
Info-Zip unzip [1 2]	libtasn1 [1 2 ...]	OpenBSD pfctl [1]
NetBSD bpf [1]	man & mandoc [1 2 3 4 5 ...]	IDA Pro [reported by authors]

Why it’s worth to take a look?

- Proven to be great for discovering security issues
- More code coverage
- No bias :)
- Long-term - cheap
  - If fuzzing is a part of development process
  - Otherwise costs for discovering every next defect increases exponentially
- Might be a great compliment to unit-testing
  - Applicable to in-process fuzzers
  - Some languages has fuzzing as a part of built-in toolchain

Coverage guided fuzzing

Takes a “graybox” or “whitebox” approaches and uses program instrumentation to trace coverage and adjust input generation to advance as far as possible.

Usually mutation based.

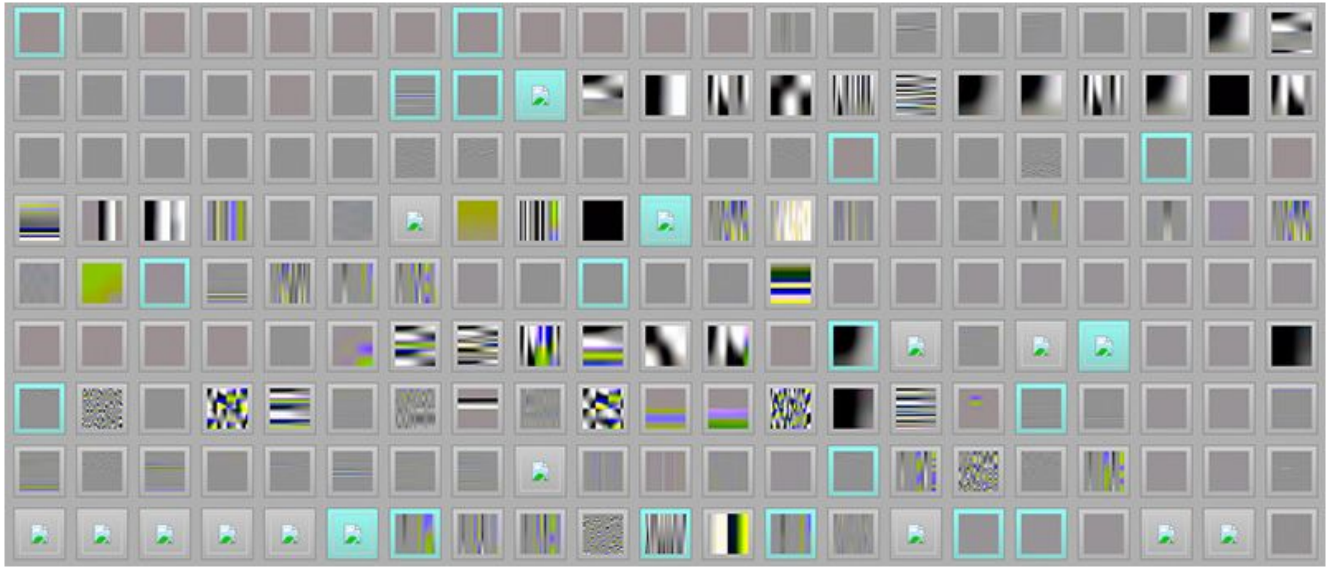
Program under test have to be instrumented additionally

Can be scarily effective on some applications. For example, pull jpeg out of the air!

```
Instrument program for a code coverage

Collect initial corpus of inputs

for {

    Mutate an input from corpus

    Execute and collect coverage

    if inputs gives new coverage {

        Add the input to corpus

    }

}
```

Pulling JPEGs out of thin air

The first image, hit after about six hours on an 8-core system, looks very unassuming: it's a blank grayscale image, 3 pixels wide and 784 pixels tall. But the moment it is discovered, the fuzzer starts using the image as a seed - rapidly producing a wide array of more interesting pics for every new execution path.



Worth to mention fuzzers and related software

- AFL
  - Works with binary executables
  - Development stopped, but still widely used
- AFLplusplus
  - Fork of AFL, seems to be the most popular one
- Gofuzz
  - If you are doing golang :)
  - Part of standard toolchain from 1.18!
- Hypofuzz
  - For python applications
- Jazzer
  - In-process fuzzer for JVM based apps
- syzkaller
  - Fuzzer for OS kernels
- libFuzzer
  - In-process fuzzer engine
  - Part of LLVM
- OSS-Fuzz
  - Continuous fuzzing of open source software
- ClusterFuzz
  - Backend for OSS-fuzz, can be run on-prem

Property based testing

Similar to fuzzing, but this term generalizes it a little bit in a sense how to define if generated testcase failed or not.

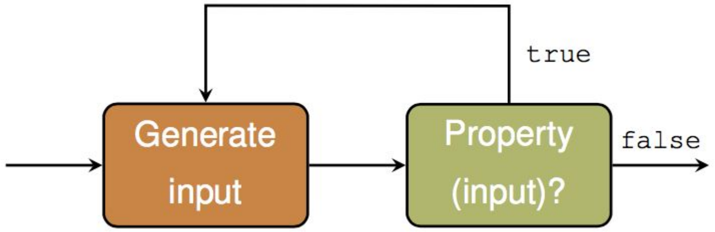
Usually describes as “the thing what QuickCheck does”. :)

Relies on “properties”. General check is - function, program or whatever system under test **abids a property despite of an input**.

Properties

Essentially:

Assertions which should be fulfilled on ***any*** provided input



Examples:

- Commutativity on the sum of ints
- Equality of algorithm output to the reference implementation
    - Porting some complex algorithm to another language
- “Round-trip”
    - compress -> decompress -> compress
    - crypt -> decrypt -> crypt
    - etc

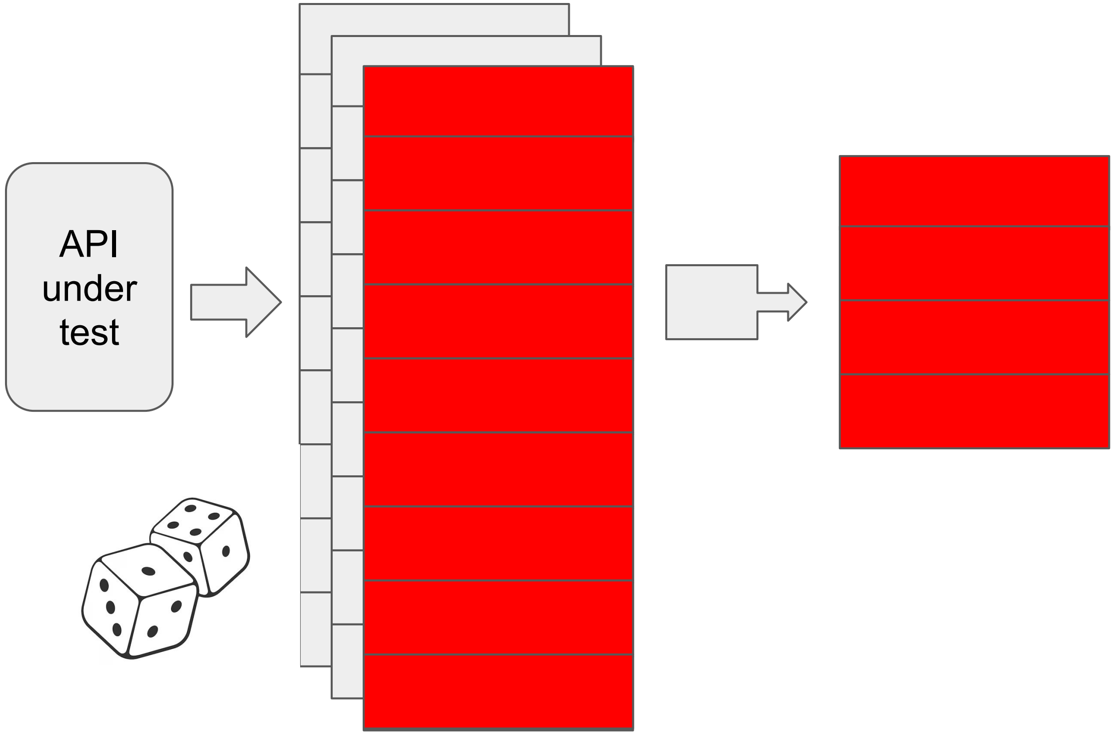
QuickCheck

- “Grandpa” of all property-based testing frameworks
- Haskell library
- Provides interface for defining data generators and properties
- Heavily relies on Haskell type system in data generation
- A lot of attempts to implement it on other languages
- Paper

Shrinking

In other words - process of finding a minimal failing example

Also called *minimization*



Hypothesis

Python framework for property based testing

- Structure aware black box fuzzer
- Has a concept of “Strategies”
  - *parser-combinators* designed for structured fuzzing
  - *parser-combinators* - parsing PRNG into a valid data structure

```python
# Our tested class
class Product:
    def __init__(self, price: float) -> None:
        self.price: float = price

    def get_discount_price(self, discount_percentage: float):
        return self.price * (discount_percentage / 100)


# The @given decorator generates examples for us!
@given(
    price=st.floats(min_value=0, allow_nan=False, allow_infinity=False),
    discount_percentage=st.floats(
        min_value=0, max_value=100, allow_nan=False, allow_infinity=False
    ),
)
def test_a_discounted_price_is_not_higher_than_the_original_price(
    price, discount_percentage
):
    product = Product(price)
    assert product.get_discount_price(discount_percentage) <= product.price
```

Libs and frameworks for property based testing

Some examples of libraries which explicitly talks about property based testing

- QuickCheck
    - Haskell
- Proper
    - Erlang
- Scala-check
    - Scala/Java
- Fast-check
    - Javascript
- Hypothesis
    - Python
- testing/quick
    - Golang, quite basic. Better use fuzzer from standard toolchain :)

Challenges adopting fuzzing and property based testing

- Setup might require a lot of effort
  - Program under test requires instrumentation
  - Fuzzers are quite complex software, requires to dive in a bit
- Side effects might be disruptive
  - Program under test MUST be somewhat isolated
- Result analysis
  - No so easy, bugreports needs deduplication
- Approach to tests writing is slightly different, assertions definitions are not that easy
- Hardly applicable to complex systems with a lot of side effects
  - Such systems needs to be splitted to separate modules
  - Challenges with tracking down and cleanup results of side effects

More materials

- The Fuzzing Book
  - Massive and comprehensive material on fuzzing overall with A LOT of examples and exercises
- google/fuzzing
  - Github repo with collection of reading materials
- Hypofuzz docs
  - Great summary of related literature and researches
- Research about reliability of UNIX utilities
  - Here “fuzzing” term was introduced
- QuickCheck original paper
  - Foundation for property-based testing concepts
- Good talk about Erlang QuickCheck
  - Covers some real world examples and usage of property based testing in the field

Showcase

Gofuzz demo.

Code is living in github repo: https://github.com/lobziik/fuzzn-meetup

Conclusion on fuzzing and property based testing

- Using such techniques might be beneficial for Dev and QA/QE side of the house
  - On a different levels of integration of course
- Great complementation to unit and integration tests
  - Covers way more than classic examples based tests
  - Can reveal bugs and unexpected behaviour far earlier
- A lot of tools already exists, just pick suitable and use it
  - ClusterFuzz along with supported fuzzers might be a great addition to your CI systems

That's all Folks!