Greybox Fuzzing

Shin Hong
CSEE Handong Global Univ.

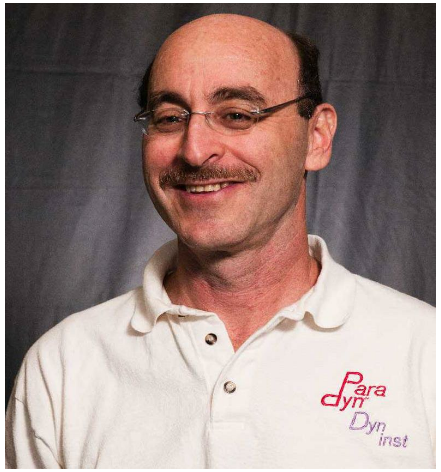
Topics

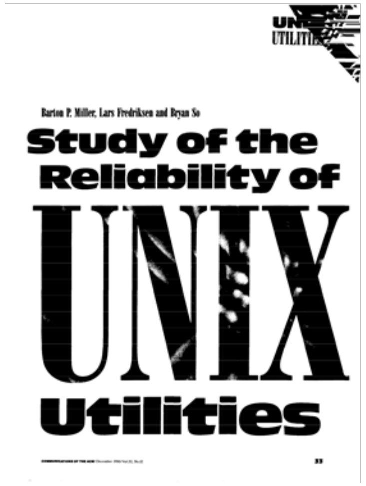
- fuzzing background
  - mutation-based fuzzing
  - greybox fuzzing
- introduction to the libFuzzer tool
  - functionalities
  - tool structure
  - walkthrough example
- engineering aspects of unit test fuzzing

It was a Dark and Stormy Night in the Fall of 1988

http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html

- Barton Miller, a professor of U. Wisconsin-Madison experienced that UNIX systems crashed extraordinary frequently.
- He conjected that it was because unexpectedly strong electric noise induced multiple tweaks in packets
- To test his conjecture, Miller gave an assignment to students to test UNIX utilities by feeding intentionally randomized inputs
  - Miller et al., An empirical study of the reliability of UNIX utilities, CACM, 1990

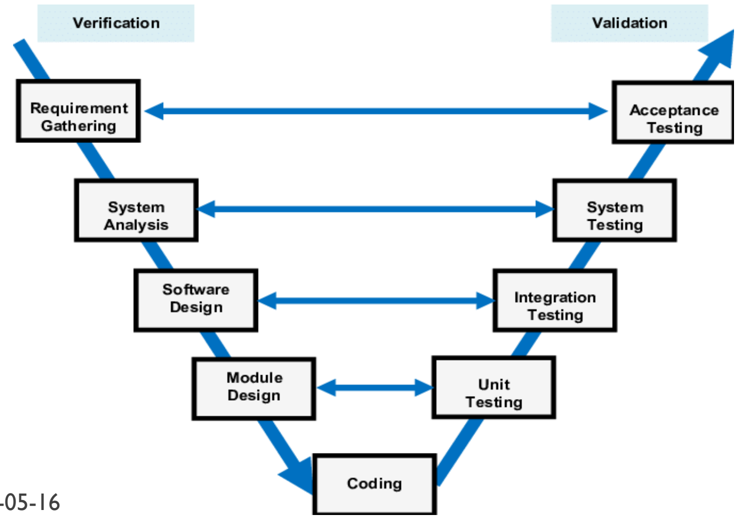


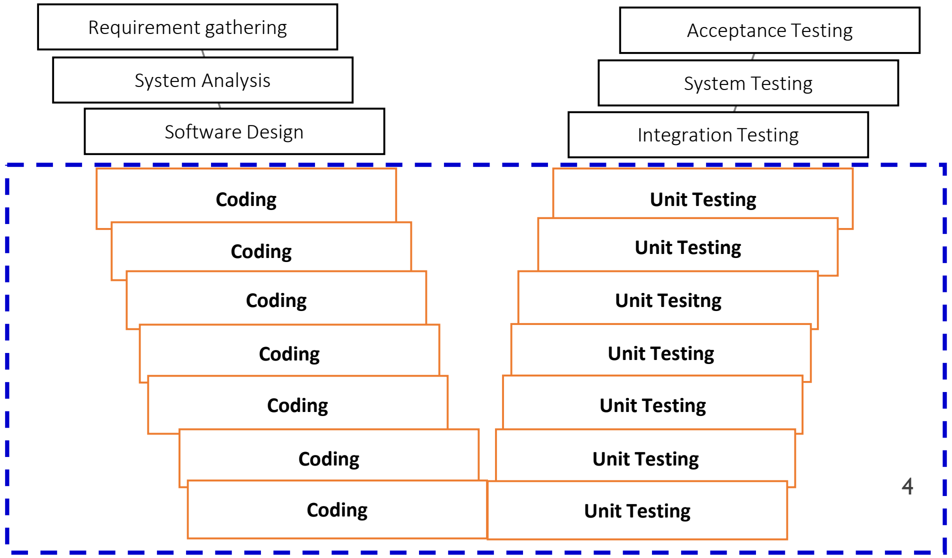

Barton P. Miller, Lars Fredriksen and Bryan So

Study of the Reliability of UNIX Utilities

Ancient Fuzzers

- Generate a long sequence of random texts that have similar aspects as formatted text input for testing UNIX command utilities
  - intermix comma, semicolon, and many control characters
    - e.g., `'!7#%"*#0=)$;%6*;>638:*>80"=</>(/*:-(2<4 !:5*6856&?""11<7+%<%7,4.8`
  - Feed randomly generated texts to a target UNIX utility, and repeat this for many hours
- By using this kind of ancient fuzzers, new bugs were found from one third of the UNIX utilities





Shortcomings of Ancient Fuzzers

- Ancient fuzzers detect only crashes and hangs, but cannot uncover silent illegal behaviors which can result much critical consequences
  - reliability issue ⇒ security issue (adversarial users)
  - employ dynamic analyzers to detect and/or predict silent violations
    - e.g., valgrind, electric fence, LLVM sanitizer suites (AddressSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer)

- Randomly generated inputs cover only restricted portion of the source code
  - random inputs are often rejected quickly because they likely have trivial input grammar errors
  - extremely low probability for a randomly generated text to pass grammar checks

Mutation-based Fuzzing

- Ideas
  - start with a set of valid inputs (*seeds*)
  - repeatedly introduce small changes to the existing inputs (*mutation*) with a hope that they exercise new behaviors

- Example: fuzzing a URL parsing libary
  - Seed
    - http://www.google.com/search?q=fuzzing
  - Fuzzed inputs
    - http://www.g=oNogl.om/search?q=fuzzing/
    - RttpX://w)ww.goo(gle.comq/sarc(q=fuzzng
    - hdt8p://"wWw.goole.com/seDarb`*?q=fuzzing
    - hup://www.google.comC/search?q=fuzzing
    - http://w7w.google.com/search?q=ufuzgzing
    - http://w&ww.google.cKom/search7q=fuzzing

Mutation Operators

- Flip one random bit
- Alternate one or multiple consecutive bytes
- Erase one or multiple bytes from random offsets
- Insert one or multiple bytes to random offsets
- Repeat existing bytes multiple times
- Add a word from a predefined dictionary
- Shuffle consecutive bytes (reorder multiple bytes randomly)
- Copy a substring and paste it randomly offsets
- Crossover
- Apply mutation one or more times on a single seed input

Fine-grained

Coarse-grained

Why Mutation Effectively Disclose Subtle Behaviors?

- It is likely to obtain quality seed inputs from existing test cases

- An error-revealing input mostly resides close to a valid input
  - close in lexical distance, or numerical distance
  - competent programmer hypothesis

- A part of a program input is likely associated with only few program components
  - an aspect of an input text can be represented as a short subsequence
  - strong locality exists in a well-modularized program

- A critical value of a specific part of input is likely found in the other parts of the inputs

Greybox Fuzzing: Use Structural Coverage to Guide Fuzzing

- Idea
  - Start with a set of valid inputs
  - Repeatedly introduce small changes to the existing inputs while expecting they exercise new behaviors
  - Include the mutated input as a seed only if it explores a new behavior
    - covering a new structural test requirement

- Greybox fuzzers (e.g., AFL, libFuzzer) show in practice that use of structural coverage dramatically improves effectiveness of mutation-based fuzzing
  - Google runs fuzzing on 160 open-source projects with 250,000 machines
  - Google found more than 16,000 bugs in Chrome by fuzzing

Basic Algorithm

Input: a target program *Prog*
a set of seeds $S = \{s_1, s_2, \ldots s_n\}$
Output: two sets of tests $P = \{p_1, p_2, \ldots, p_m\}$, $F=\{f_1, f_2, \ldots, f_k\}$
Procedure:

$P \leftarrow S$, $F \leftarrow \emptyset$, $C \leftarrow \emptyset$
while $p \in P$ **begin**
    $C \leftarrow C \cup \mathrm{Cov}(Prog, p)$
end while
while termination condition is not satisfied **begin**
    $p \leftarrow$ select a random test input from P
    $p' \leftarrow$ mutate p with a certain mutation operator
    if $Prog(p')$ fails **then**
        $F \leftarrow F \cup \{ p' \}$
    else
        if $\mathrm{Cov}(Prog, p') - C \neq \emptyset$ **then**
            $P \leftarrow P \cup \{ p' \}$
            $C \leftarrow C \cup \mathrm{Cov}(Prog, p')$
        end if
    end if
end while

libFuzzer: Fuzzing Tool for LLVM

https://llvm.org/docs/LibFuzzer.html

- libFuzzer is a greybox fuzzer inspired by AFL for testing C/C++ libraries
  - developed as a component of LLVM
    - target C/C++ programs
    - well integrated with the LLVM sanitizer suites
  - generate inputs to public APIs in a unit test driver (rather than a system input)
  - provide a plugin API for defining and managing custom mutation operators
    - easy to implement structure-aware, grammar-based fuzzing
- libFuzzer, together with AFL, is used as a core component of OSS-Fuzz and ClusterFuzz https://google.github.io/clusterfuzz/

libFuzzer Mutation Operators

Mutator	Description
EraseBytes	Reduce size by removing a random byte
InsertByte	Increase size by one random byte
InsertRepeated Bytes	Increase size by adding at least 3 random bytes
ChangeBit	Flip a Random bit
ChangeByte	Replace byte with random one
ShuffleBytes	Randomly rearrange input bytes
ChangeASCII Integer	Find ASCII integer in data, perform random math ops and overwrite into input.
ChangeBinary Integer	Find Binary integer in data, perform random math ops and overwrite into input
CopyPart	Return part of the input
CrossOver	Recombine with random part of corpus/self
AddWordPersist AutoDict	Replace part of input with one that previously increased coverage (entire run)
AddWordTemp AutoDict	Replace part of the input with one that recently increased coverage
AddWord FromTORC	Replace part of input with a recently performed comparison

- Domain-specific word dictionary can be configured for a specific target function
- We can add custom mutation operators
  - alternate an input text considering its grammar or constraints on input validity

Writing Unit Fuzzing Driver (parameterized unit test case)

- *target function* accepts array of bytes, and feed accepted data into the API under test

```
// target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    DoSomethingInterestingWithMyAPI(Data, Size);
    return 0; // Non-zero return values are reserved for future use.
}
```

- aspects
  - set prerequisite environment to run target API
    - configure test execution environment
    - invoke other APIs to set the starting state and also mock objects
  - cast given fuzzed input to the arguments of a target API
    - typecasting (e.g., a region of string to an integer)
    - precondition checking
    - selecting sub-cases of a test scenario
  - configure fuzzing engine

Example - Triangle

- Fuzzing target

```
extern int LLVMFuzzerTestOneInput (const uint8_t *Data, size_t Size) {
        if (Size != 12)
                return 0 ;
        int a, b, c ;
        a = *((int *) (Data + 0)) ;  b = *((int *) (Data + 4)) ;  c = *((int *) (Data + 8)) ;

        triangle_type(a, b, c) ;
}
```

- Commands

```
clang -g -O1  -fsanitize=fuzzer,signed-integer-overflow triangle.c fuzz_target.c -o test-target
export UBSAN_OPTIONS=halt_on_error=1
./test-target corpus seed -max_len=100
```

References

The Fuzzing Book: Tools and Techniques for Generating Software Tests

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holle

https://www.fuzzingbook.org/

libFuzzer – a library for coverage-guided fuzz testing

https://llvm.org/docs/LibFuzzer.html

The Art, Science and Engineering of Fuzzing: A Survey

V. J.M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo

IEEE Transactions on Software Engineering, Volume: 47, Issue: 11, 01 November 2021