Unit Fuzz Testing for C/C++ Programs

KCSE 2021 Tutorial

Shin Hong Hanyoung Yoo

2 Feburary 2021

ARISE
Advancing Reliability, Safety
& Security of Software Engineering
https://arise.handong.edu

HGU 한동대학교
HANDONG GLOBAL UNIVERSITY

ARISE: Advancing Reliability, Safety & Security in SE @ Handong

https://arise.handong.edu/

- Research goal
  - study the phenomena and the principles of software developments
  - find better ways of constructing reliable, secure, and safe software
  - develop automated debugging and testing techniques

- Research interests
  - test generation
  - automated debugging
  - static and dynamic analyses

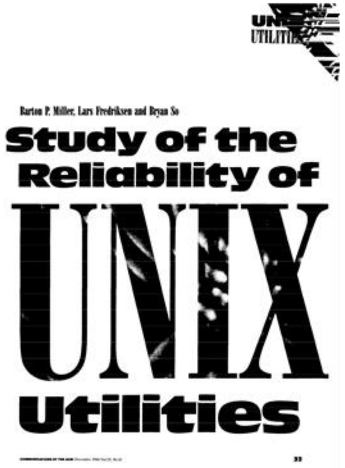
In this tutorial, we will discuss

- 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

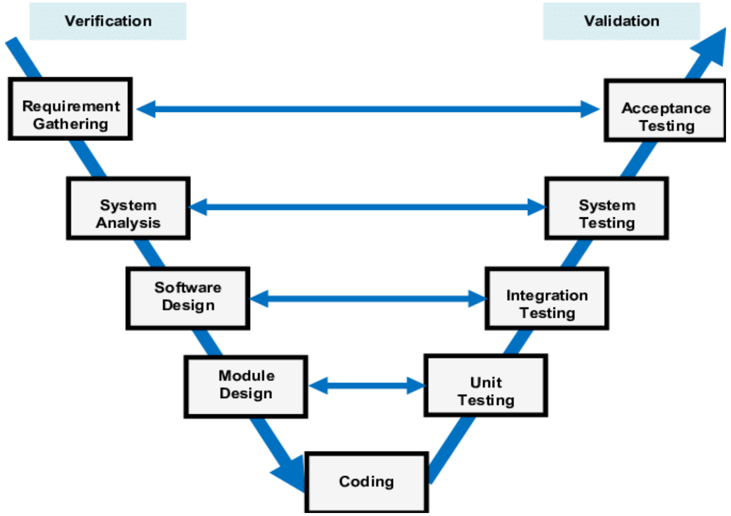
http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html

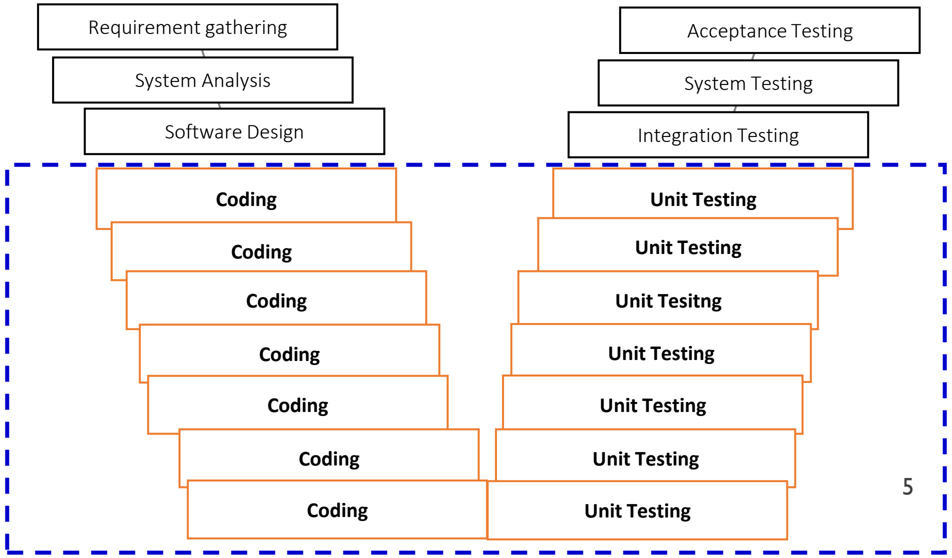
- Barton Miller, a professor of U. Wisconsin-Madison experienced that UNIX systems crashed extraordinary frequently.
- He conjected that it was because unexpectedly strong electric noise induced multiple tweaks in packets
- To test his conjecture, Miller gave an assignment to students to test UNIX utilities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



Anceint 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 250000 machines
  - Google found more than 16,000 bugs in Chrome have been found by fuzzing

Basic Algorithm

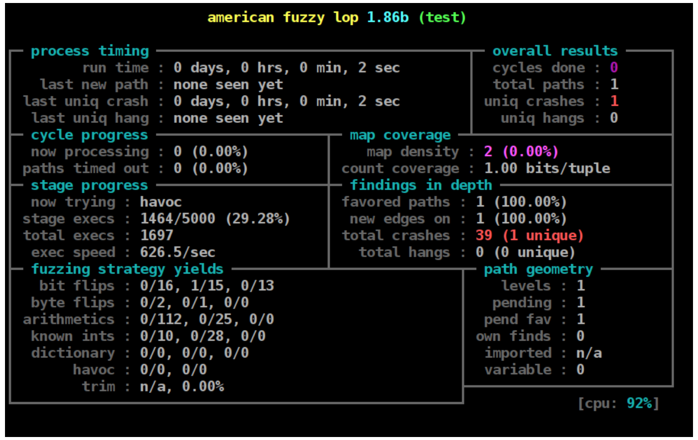
Input: a target program *Prog*
a set of seeds $S = \{s_1, s_2, \dots s_n\}$
Output: two sets of tests $P = \{p_1, p_2, \dots, p_m\}$, $F=\{f_1, f_2, \dots, f_k\}$
Procedure:

$P \leftarrow S$, $F \leftarrow \emptyset$, $C \leftarrow \emptyset$
while $p \in P$ **begin**
    $C \leftarrow C \cup \text{Cov}(Prog, p)$
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 $\text{Cov}(Prog, p') - C \neq \emptyset$ **then**
            $P \leftarrow P \cup \{ p' \}$
            $C \leftarrow C \cup \text{Cov}(Prog, p')$
        end if
    end if
end while

American Fuzzy Lop (AFL)

- AFL is an open-source fuzzer that employs genetic algorithms to efficiently increase code coverage of the test cases.
  - Used for detecting significant software bugs in major free software projects such as OpenSSL, Firefox, SQLite, etc.

- AFL is a dumb mutation-based grey-box fuzzer that collects coverage information on the basic block transitions that are exercised by an input
  - The block transition coverage (i.e., branch coverage), along with coarse-grained branch-taken hit counts, is a set of pairs of the form $(branch\ ID, branch\ hits)$
  - AFL addresses path explosion by bucketing
    - Divides into several buckets by considering coarse tuple hit counts
      {1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+}
    - Changes within the range of a single bucket are ignored.



Algorithm

```
procedure FuzzTest(Prog, Seeds)
 Queue ← Seeds
 while not Terminate() do
    for input in Queue do
      score ← PerformanceScore(Prog, input)
      for offset ∈ {0, 1, 2, … , |input| − 1} do
        for m ∈ DeterministicMutationTypes do
          input′ ← Mutate(input, m, offset)
          RunAndSave(Prog, input′, Queue)
        end for
      end for
      for j ∈ {1, 2, … , score} do
        input′ ← MutateHavoc(input)
        RunAndSave(Prog, input′, Queue)
      end for
    end for
 end while
```

```
procedure RunAndSave(Prog, input, Queue)
  result ← Run(Prog, input)
  if HasNewCoverage(result) then
        AddToQueue(input, Queue)
  end if
```

```
procedure MutateHavoc(Prog, input)
  n ← Random(256)
  input′ ← input
  for k ∈ {0, 1, … , n − 1} do
      m ← RandomMutateType
      offset ← Random(|input′|)
      input′ ← Mutate(input′, m, offset)
  end for
return newinput
```

Example (1/11)

```c
void my_echo (char *data) {
  char buf[10] ;
  strcpy (buf, data) ;
  printf (“%s\n”, buf) ;
  if (data[0] == 'f')
     if (data[1] == 'u')
        if (data[2] == 'z')
           if (data[3] == 'z')
              assert (0) ;
}

int main (void) {
  char inp[50] ;
  read (STDIN_FILENO, inp, 50) ;
  my_echo (inp) ;
}
```

Testcase1 : hello

```
procedure FuzzTest(Prog, Seeds)
 Queue ← Seeds
 while not Terminate()  do
   for input in Queue do
     score ← PerformanceScore(Prog, input)
     for offset ∈ {0, 1, 2, … , |input| − 1} do
       for m ∈ DeterministicMutationTypes do
         input′ ← Mutate(input, m, offset)
         RunAndSave(Prog, input′, Queue)
       end for
     end for
     for j ∈ {1, 2, … ,  score} do
       input′ ← MutateHavoc(input)
       RunAndSave(Prog, input′, Queue)
     end for
   end for
 end while
```

Example (2/11)

$Queue = \{ \text{“hello”} \}$
$input = \text{“hello”}$
$score = 100$

```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while not Terminate() do
     for input in Queue do
       score ← PerformanceScore(Prog, input)
       for offset ∈ {0, 1, 2, … , |input| − 1} do
          for m ∈ DeterministicMutationTypes do
             input′ ← Mutate(input, m, offset)
             RunAndSave(Prog, input′, Queue)
          end for
       end for
       for j ∈ {1, 2, … , score} do
          input′ ← MutateHavoc(input)
          RunAndSave(Prog, input′, Queue)
       end for
     end for
  end while
```

PeformanceScore

- assign a higher score to an input that uncover more unseen execution features

Example (3/11)

$Queue = \{$ "hello" $\}$
$input =$ "hello"
$score = 100$

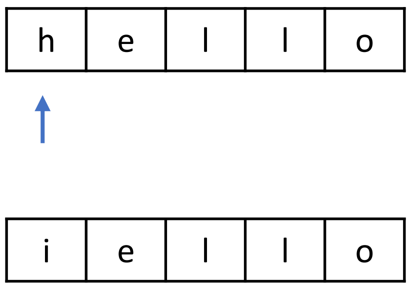
$DeterministicMutationTypes$

- bit/byte flipping
  - 1,2,4 bit/byte flipping
- arithmetic increment/decrement of integer
- combine the first half of one input with the second half of another (i.e., splicing)

```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while not Terminate() do
    for input in Queue do
      score ← PerformanceScore(Prog, input)
      for offset ∈ {0, 1, 2, … , |input| − 1} do
        for m ∈ DeterministicMutationTypes do
          input′ ← Mutate(input, m, offset)
          RunAndSave(Prog, input′, Queue)
        end for
      end for
      for j ∈ {1, 2, … , score} do
        input′ ← MutateHavoc(input)
        RunAndSave(Prog, input′, Queue)
      end for
    end for
  end while
```

Example (4/11)

$Queue$ = { “hello” }
$input$ = “hello”
$score = 100$
$i = 0$

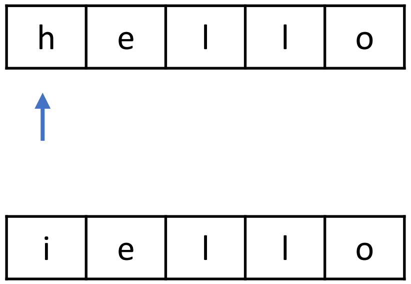



```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while not Terminate() do
    for input in Queue do
      score ← PerformanceScore(Prog, input)
      for offset ∈ {0, 1, 2, … , |input| − 1} do
        for m ∈ DeterministicMutationTypes do
          input′ ← Mutate(input, m, offset)
          RunAndSave(Prog, input′, Queue)
        end for
      end for
      for j ∈ {1, 2, … , score} do
        input′ ← MutateHavoc(input)
        RunAndSave(Prog, input′, Queue)
      end for
    end for
  end while
```

```
procedure RunAndSave(Prog, input , Queue)
  results ← RUN(Prog, input)
  if HasNewCoverage(results) then
      AddToQueue(input, Queue)
  end if
```

Example (5/11)

$Queue = \{ \text{“hello”} \}$
$input = \text{“hello”}$
$score = 100$
$i = 0$



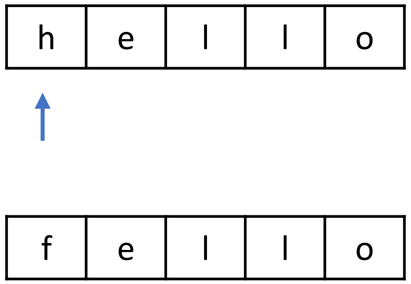

```
procedure RunAndSave(Prog, input , Queue)
  results ← RUN(Prog, input)
  if HasNewCoverage(results) then
      AddToQueue(input, Queue)
  end if
```

```c
void my_echo (char *data) {
  char buf[10] ;
  strcpy (buf, data) ;
  printf (“%s\n”, buf) ;
  if (data[0] == 'f')
     if (data[1] == 'u')
        if (data[2] == 'z')
           if (data[3] == 'z')
              assert (0) ;
}

int main (void) {
  char inp[50] ;
  read (STDIN_FILENO, inp, 50) ;
  my_echo (inp) ;
}
```

Example (6/11)

$Queue = \{ \text{"hello"} \}$
$input = \text{"hello"}$
$score = 100$
$i = 0$




```
procedure RunAndSave(Prog, input, Queue)
  results ← RUN(Prog, input)
  if HasNewCoverage(results) then
        AddToQueue(input, Queue)
  end if
```

```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while not Terminate() do
    for input in Queue do
      score ← PerformanceScore(Prog, input)
      for offset ∈ {0, 1, 2, … , |input| − 1} do
        for m ∈ DeterministicMutationTypes do
          input′ ← Mutate(input, m, offset)
          RunAndSave(Prog, input′, Queue)
        end for
      end for
      for j ∈ {1, 2, … , score} do
        input′ ← MutateHavoc(input)
        RunAndSave(Prog, input′, Queue)
      end for
    end for
  end while
```

Example (7/11)

$Queue$ = { “hello” ,“fello” }
$input$ = “hello”
$score = 100$
$i = 0$

h	e	l	l	o

f	e	l	l	o

```
procedure RunAndSave(Prog, input , Queue)
  results ← RUN(Prog, input)
  if HasNewCoverage(results) then
        AddToQueue(input, Queue)
  end if
```

```c
void my_echo (char *data) {
  char buf[10] ;
  strcpy (buf, data) ;
  printf (“%s\n”, buf) ;
  if (data[0] == 'f')
     if (data[1] == 'u')
        if (data[2] == 'z')
           if (data[3] == 'z')
              assert (0) ;
}

int main (void) {
  char inp[50] ;
  read (STDIN_FILENO, inp, 50) ;
  my_echo (inp) ;
}
```

Example (8/11)

$Queue = \{$ "hello", "fello" $\}$
$input =$ "hello"
$score = 100$
$i = 0, \quad j = 0$

h	e	l	l	o

MutateHavoc

- insertion, deletion, arithmetics, and splicing of different test cases
- random mutate random offsets

```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while not Terminate() do
    for input in Queue do
      score ← PerformanceScore(Prog, input)
      for offset ∈ {0, 1, 2, … , |input| − 1} do
        for m ∈ DeterministicMutationTypes do
          input′ ← Mutate(input, m, offset)
          RunAndSave(Prog, input′, Queue)
        end for
      end for
      for j ∈ {1, 2, … , score} do
        input′ ← MutateHavoc(input)
        RunAndSave(Prog, input′, Queue)
      end for
    end for
  end while
```

Example (9/11)

$Queue = \{ \text{"hello"}, \text{"fello"} \}$
$input = \text{"hello"}$
$score = 100$
$i = 0, \quad j = 0, \quad k = 1$
$n = 2$
$m = Insertion$

h	e	l	^	$	l	o

h	e	<	Q	4	!	l	^	$	l	o

procedure MutateHavoc($Prog, input$)
  $n \leftarrow$ Random(256)
  $input' \leftarrow input$
  for $k \in \{0, 1, \ldots, \text{n} - 1\}$ **do**
    $m \leftarrow$ RandomMutateType
    $offset \leftarrow$ Random($|input'|$)
    $input' \leftarrow$ Mutate($input', m, offset$)
  end for
return $newinput$

procedure FuzzTest($Prog, Seeds$)
  $Queue \leftarrow Seeds$
  while not Terminate() **do**
    for $input$ **in** $Queue$ **do**
      $score \leftarrow$ PerformanceScore($Prog, input$)
      for $offset \in \{0, 1, 2, \ldots, |input| - 1\}$ **do**
        for $m \in DeterministicMutationTypes$ **do**
          $input' \leftarrow$ Mutate($input, m, offset$)
          RunAndSave($Prog, input', Queue$)
        end for
      end for
      for $j \in \{1, 2, \ldots, score\}$ **do**
        $input' \leftarrow$ MutateHavoc($input$)
        RunAndSave($Prog, input', Queue$)
      end for
    end for
  end while

Example (10/11)

$Queue = \{ \text{“hello”}, \text{“fello”} \}$
$input = \text{“hello”}$
$score = 100$
$i = 0, \quad j = 0, \quad k = 1$
$n = 2$
$m = Insertion$

h	e	<	Q	4	!	l	^	$	l	o

procedure RunAndSave(*Prog*, *input* , *Queue*)
  results ← RUN(*Prog*, *input*)
  if HasNewCoverage(*results*) **then**
    AddToQueue(*input*, *Queue*)
  end if

```
void my_echo (char *data) {
  char buf[10] ;
  strcpy (buf, data) ;
  printf ("%s\n", buf) ;
  if (data[0] == 'f')
    if (data[1] == 'u')
      if (data[2] == 'z')
        if (data[3] == 'z')
          assert (0) ;
}

int main (void) {
  char inp[50] ;
  read (STDIN_FILENO, inp, 50) ;
  my_echo (inp) ;
}
```

crash

Example (11/11)

- crashes/
  - id0 : he<Q4!l^$lo
  - id1 : fuzz

```c
void my_echo (char *data) {
  char buf[10] ;
  strcpy (buf, data) ;
  printf (“%s\n”, buf) ;
  if (data[0] == 'f')
    if (data[1] == 'u')
      if (data[2] == 'z')
        if (data[3] == 'z')
          assert (0) ;
}

int main (void) {
  char inp[50] ;
  read (STDIN_FILENO, inp, 50) ;
  my_echo (inp) ;
}
```

Favorite and Interesting Inputs

- An input t is **interesting** if t executes a path where a transition b is exercised n times and for all other inputs t' that exercise b for m times, $\lfloor \log n \rfloor \neq \lfloor \log m \rfloor$ (i.e., different buckets)

- AFL identifies an input as **favorite** if it is the fastest and smallest input for any of the control-flow edges it exercises
  - AFL mostly ignores non-favorite seeds when selecting next inputs

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)
  - run multiple test runs in a single process for fast fuzzing
  - 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: Fuzzing Algorithm

```
FuzzingLoop(Prog, Seeds) begin
  Queue ← Seeds
  while not Terminate()  do
    input  ← select an element in Queue with probability propotional to its weight
    for i ∈ {1, 2, … ,  depth} do
      offset ← random(0, |input| - 1)
      mutator  ← a random mutation operator
      input′ ← Mutate(input, offset, mutator)
      result ← Run(Prog, input)
      if HasNewCoverage(result) then
        Add(Queue,  input′)
      end if
    end for
  end while
end
```

- greater if it is added more recently
- greater if it reaches to branches that are rarely covered by other inputs

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: Target Function of SQLite3 in Google Fuzzer Test Suite

https://github.com/google/fuzzer-test-suite

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    ...
    if( size < 3 ) return 0;    /* Early out if unsufficient data */

    if( data[1]=='\n' ){
        uSelector = data[0]; data += 2; size -= 2;
    }else{
        uSelector = 0xfd;
    }
    rc = sqlite3_open_v2("fuzz.db", &db,
                         SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE | SQLITE_OPEN_MEMORY, 0);
    if( rc ) return 0;
    if( uSelector & 1 )
        sqlite3_progress_handler(db, 4, progress_handler, (void*)&progressArg);

    uSelector >>= 1;
    progressArg = uSelector & 1;  uSelector >>= 1;

    sqlite3_db_config(db, SQLITE_DBCONFIG_ENABLE_FKEY, uSelector&1, &rc);
    uSelector >>= 1;

    execCnt = uSelector + 1;
    sqlite3_exec(db, sqlite3_mprintf("%.*s", (int)size, data), exec_handler, (void*)&execCnt, &zErrMsg);

    sqlite3_free(zErrMsg);
    sqlite3_free(zSql);
    sqlite3_close(db);   }
```

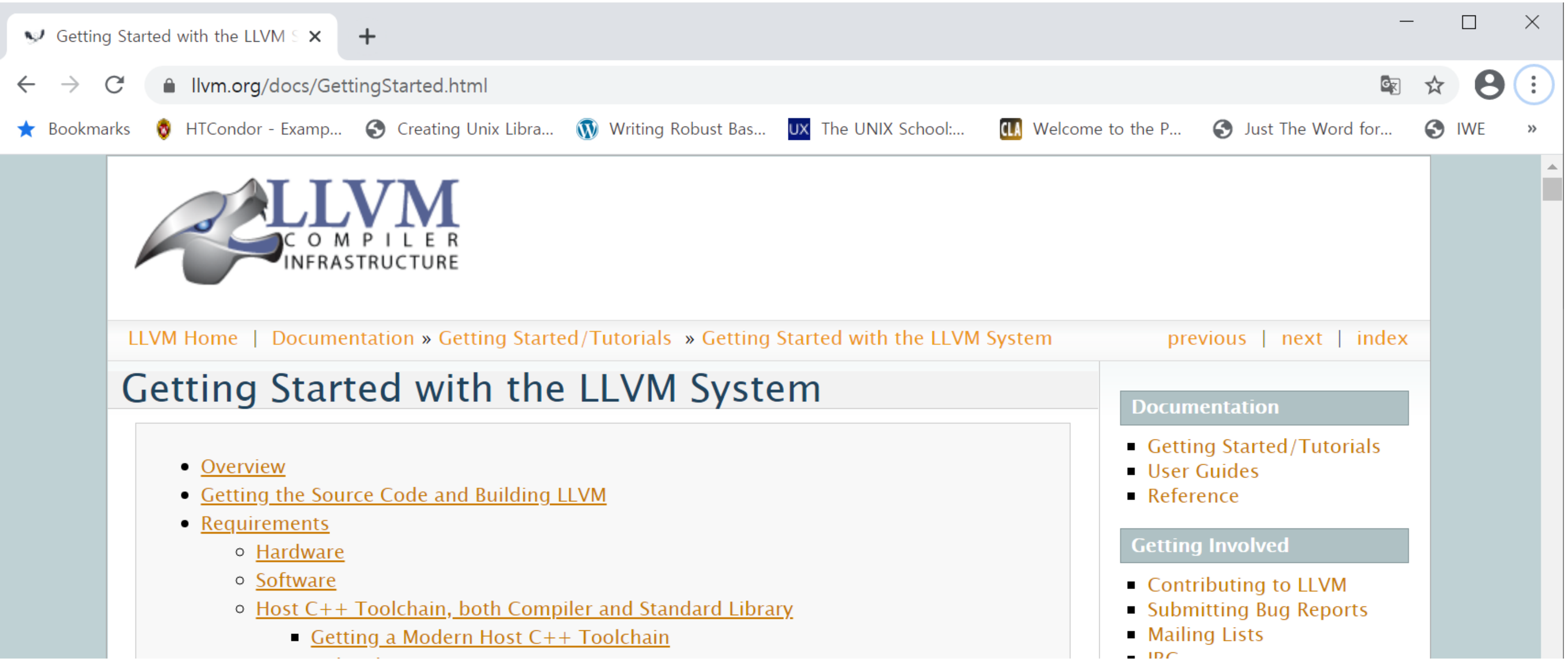
		7	6	5	4	3	2	1	0
0	b	1	1	1	1	1	1	0	1

uSelector

- Bit 0 enable or disable progress handler
- Bit 1 indicates progress handler return value
- Bit 2 enable or disable foreign key constraints
- Bits 3-7 indicates the number of result rows

Installing libFuzzer

- Download and build LLVM with `clang` and `compiler-rt`
  - install a lastest `llvm-toolset` package



Target Build

```
clang -g -O1 -fsanitize=fuzzer                         mytarget.c # Builds the fuzz target w/o sanitizers
clang -g -O1 -fsanitize=fuzzer,address                 mytarget.c # Builds the fuzz target with ASAN
clang -g -O1 -fsanitize=fuzzer,signed-integer-overflow mytarget.c # Builds the fuzz target with a part of UBSAN
clang -g -O1 -fsanitize=fuzzer,memory                  mytarget.c # Builds the fuzz target with MSAN
```

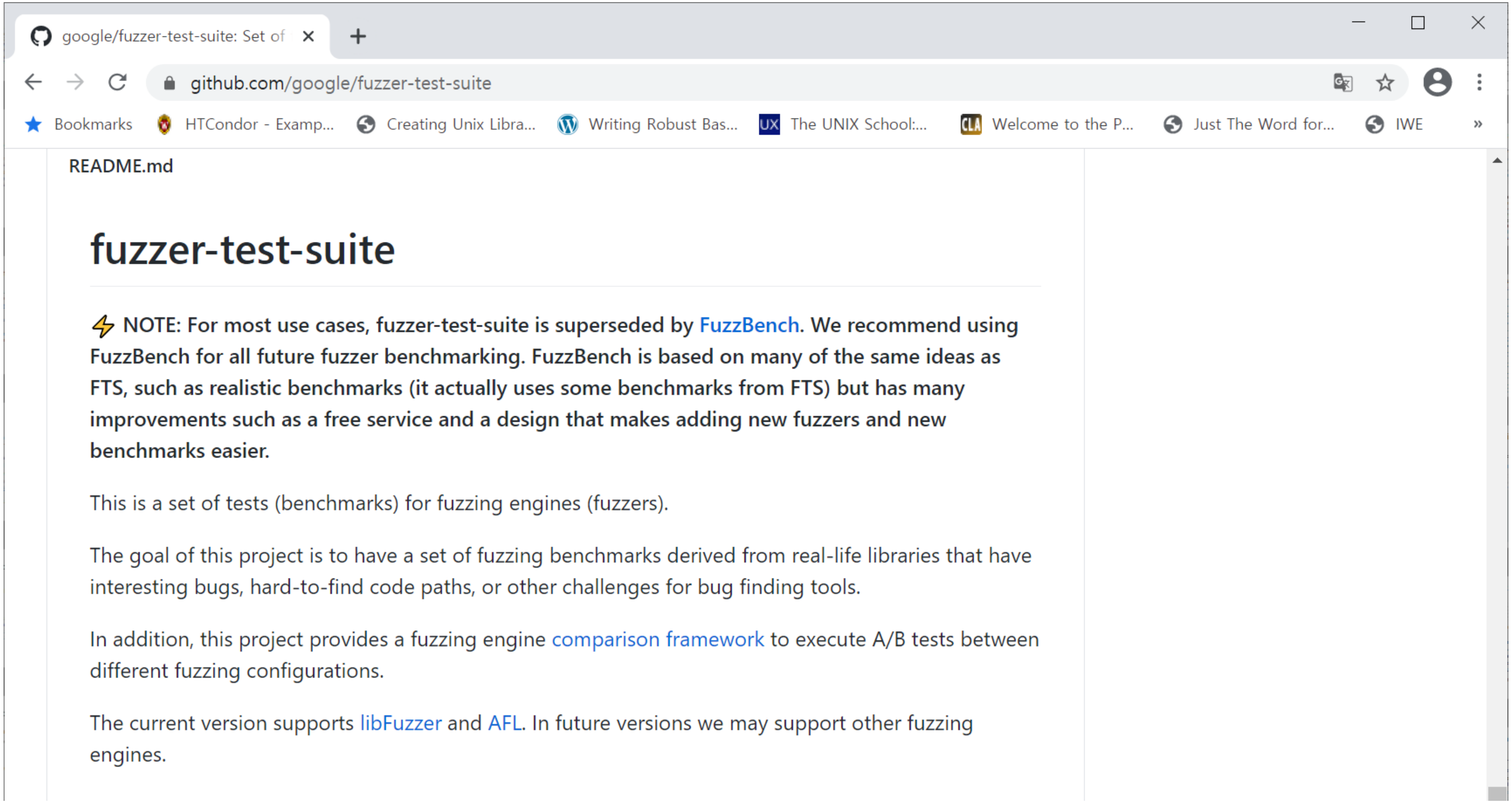
- Build the target project and fuzzing driver using Clang with fuzzer option
  - IR-level instrumentation is made for runtime tracing and fuzzing
  - it is possible to turn on LLVM santifizer runtime checkers
  - more build options
    - `-fsanitize=fuzzer-no-link` : request just the instrumentation without linking
    - `-fno-omit-frame-pointer` : get nicer stack straces in error messages
    - `-gline-tables-only` : enables debug info, makes the error messages easier to read
- An executable fuzzing driver is created as result

Fuzzing Driver Execution

```
$ ./fuzz_driver [Options] <CorpusDir> <SeedDir>
```

- **Generated Input corpus**: fuzzer stores all interesting program inputs as result of an execution
- **Seed inputs**: user can provide the initial set of inputs, or the fuzzer starts with an empty string as an initial input.
- **Options**
  - `-runs` : the number of individual test runs, -1(the default) to run indefinitely
  - `-max_total_time` : the maximum total time in seconds to run the fuzzer
  - `-max_len` : maximum length of a test input. If 0(the default), libFuzzer tries to guess a good value based on the corpus (and reports it).
  - `-timeout` : timeout in seconds, default 1200. If an input takes longer than this timeout, the process is treated as a failure case
  - `-workers=N` : run fuzzer with N processes concurrently
  - `-dict=filename` : load the keywords in the given file to the fuzzer dictionary

Demo: libxml of Google Fuzz Test Suite



fuzzer-test-suite

NOTE: For most use cases, fuzzer-test-suite is superseded by FuzzBench. We recommend using FuzzBench for all future fuzzer benchmarking. FuzzBench is based on many of the same ideas as FTS, such as realistic benchmarks (it actually uses some benchmarks from FTS) but has many improvements such as a free service and a design that makes adding new fuzzers and new benchmarks easier.

This is a set of tests (benchmarks) for fuzzing engines (fuzzers).

The goal of this project is to have a set of fuzzing benchmarks derived from real-life libraries that have interesting bugs, hard-to-find code paths, or other challenges for bug finding tools.

In addition, this project provides a fuzzing engine comparison framework to execute A/B tests between different fuzzing configurations.

The current version supports libFuzzer and AFL. In future versions we may support other fuzzing engines.

Aspects of Fuzz Test Engineering (1/2)

- Constructing unit test driver (target function)
  - environment set up
  - arguments to be randomized
  - size of each argument
  - test oracle

- Mutation operators
  - selecting built-in mutation operators
  - domain-specific dictionary
  - customized mutation operators considering domain-specific input characteristics

Aspects of Fuzz Test Engineering (2/2)

- Seed Input
  - selecting/distilling regression test cases
  - generating seeds from given input grammar

- Engine configuration
  - fuzzing time
  - seed scheduling algorithm
  - use of data-flow-sensitive mutation
  - use of dynamic checkers

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/

American fuzzy lop

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

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, Early Access, https://arxiv.org/abs/1812.00140