SM2-TES: Functional Programming and Property-Based Testing, Day 10

Jan Midtgaard

MMMI, SDU



Motivation Intermezzo: Challenges in C Programming Fuzz Testing Fuzzing in the Bigger Picture



Motivation



Positive vs. negative testing

QuickCheck (as we've covered so far) is focused mainly on positive testing: Testing that systems/programs/libraries act as desired on valid input.

These days it is increasingly important to also perform **negative testing**: Testing that systems/programs/libraries also act meaningfully on invalid input.

SQL or JS injections, buffer overflows, ...

Interestingly, a randomized testing approach termed fuzz testing (or fuzzing) has evolved in parallel among working software engineers.

SDU 🎓

Intermezzo: Challenges in C Programming



The challenges of C programming...

Writing proper C code is challenging:

- writing out-of-bounds in the heap or stack
- reading out-of-bounds in the heap or stack
- uninitialized reads
- □ use-after-free
- double freeing

...

Historically compilers did not help catch these.

With a crash you would be lucky...

More likely: program sometimes exhibits weird behavior SDU 🎓 Google software engineers at some point designed AddressSanitizer (ASan) Which was a game changer.

- https://github.com/google/sanitizers/wiki/AddressSanitizer
- ASan's job is simply to detects memory errors
- In a language like Java we're spoiled and take, e.g., IndexOutOfBounds exceptions, for granted.
- These days LLVM (and GCC) has ASan support built in.
- ASan currently supports Linux, *BSD, Mac

Historically, a range of tools predated ASan: Valgrind, Electric Fence, ...

SDU 🎓

An example C program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  char *p = malloc(5);
  p[0] = 'h';
 p[1] = 'e';
 p[2] = 'l';
 p[3] = '1';
 p[4] = ' o';
  p[5] = ' \setminus 0'; /* this index is out of bounds */
  printf("%s\n",p);
  return 0;
}
```

An example C program

```
#include <stdio.h>
 #include <stdlib.h>
 int main()
 {
   char *p = malloc(5);
   p[0] = 'h';
   p[1] = 'e';
   p[2] = 'l';
   p[3] = 'l';
   p[4] = ' \circ';
   p[5] = ' \setminus 0'; /* this index is out of bounds */
   printf("%s\n",p);
   return 0;
 }
$ clang -Wall -Wextra -pedantic -o example2 example2.c
$ ./example2
hello
             Hm, it runs without any flags raised...
$
```

Same example compiled with ASan

\$ clang -Wall -Wextra -pedantic -o example2 -fsanitize=address example2.c
\$

Same example compiled with ASan

\$ clang -Wall -Wextra -pedantic -o example2 -fsanitize=address example2.c
\$./example2

==56166==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000b5 at pc 0x00010faa7df5 bp 0x7fff50158990 sp 0x7fff50158988 WRITE of size 1 at 0x602000000b5 thread T0

- #0 0x10faa7df4 in main (example2:x86_64+0x100000df4)
- #1 0x7fffad83f234 in start (libdyld.dylib:x86_64+0x5234)

0x602000000b5 is located 0 bytes to the right of 5-byte region [0x60200000b0,0x60200000b5) allocated by thread T0 here:

- #0 0x10fb04e9c in wrap_malloc (libclang_rt.asan_osx_dynamic.dylib:x86_64h+0x58e9c)
- #1 0x10faa7bfa in main (example2:x86_64+0x100000bfa)
- #2 0x7fffad83f234 in start (libdyld.dylib:x86_64+0x5234)

```
SUMMARY: AddressSanitizer: heap-buffer-overflow (example2:x86_64+0x100000df4) in main Shadow bytes around the buggy address:
```

[...]

==56166==ABORTING Abort trap: 6



ASan doesn't catch everything

For example:

SDU 🎸

ASan doesn't catch everything

For example:

ASan doesn't catch everything

For example:

This motivated the development of another tool: MemorySanitizer (MSan)

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

SDU SDU MSan only supports Linux x86_64 (for now) 10/34

- C and C++ have other causes of undefined behavior, e.g.:
- misaligned or null pointers
- □ integer overflow
- floating-point conversion overflow

UndefinedBehaviorSanitizer (UBSan) can catch these.

http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html



- C and C++ have other causes of undefined behavior, e.g.:
- misaligned or null pointers
- □ integer overflow
- floating-point conversion overflow

UndefinedBehaviorSanitizer (UBSan) can catch these.

http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

- C and C++ have other causes of undefined behavior, e.g.:
- misaligned or null pointers
- □ integer overflow
- floating-point conversion overflow

UndefinedBehaviorSanitizer (UBSan) can catch these.

http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

SDU 🎓

- C and C++ have other causes of undefined behavior, e.g.:
- misaligned or null pointers
- □ integer overflow
- floating-point conversion overflow

UndefinedBehaviorSanitizer (UBSan) can catch these.

http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

```
int main(int argc, char **argv) {
    int k = 2147483647; /* max int */
    k = k + argc; /* argument count is 1 without cmdline arguments */
    return 0;
    }
$ clang -o ubsan -fsanitize=undefined ubsan.c
$ ./ubsan
ubsan.c:3:9: runtime error: signed integer overflow: 2147483647 + 1
cannot be represented in type 'int'
$
```

SDU (LLVM supports UBSan on Android, Linux, BSD, and OSX)^{11/34}

ASan/MSan/UBSan and property-based testing

Bottom line:

ASan, MSan, and UBSan raise nice big red flags when something bad happens.

They make a real difference — also "just" within traditional unit testing ASan/MSan/UBSan and property-based testing

Bottom line:

ASan, MSan, and UBSan raise nice big red flags when something bad happens.

They make a real difference — also "just" within traditional unit testing

But: they also lend themselves to automated testing with the property *"program runs without raising any flags"* (as an addition to *"program runs without crashing"*). Bottom line:

ASan, MSan, and UBSan raise nice big red flags when something bad happens.

They make a real difference — also "just" within traditional unit testing

But: they also lend themselves to automated testing with the property *"program runs without raising any flags"* (as an addition to *"program runs without crashing"*).

Q: Can a buggy program still pass such tests? (hint: think devil's advocate)

SDU 🎓

Fuzz Testing



Fuzz testing, historically (1/2)

"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. It was a 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. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash ("core dump"); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us believe that there might be serious bugs lurking in the systems that we regularly used."

> From *"An Empirical Study of the Reliability of UNIX Utilities"*, Miller-Frederiksen-So, CACM'90

This experience led Miller to study how random input affected common programs.

Fuzz was the name of their random character generator.

Fuzz testing, historically (2/2)

Miller's fuzz testing research tell a fascinating tale:

- **1990:** can crash 25-33% of utility programs on UNIX
- 1995: revisiting the experiment still able to crash or hang 25-40% of basic programs and X-Window applications on a UNIX system
- **2000:** Crash 21% and hang 24% of Windows NT applications with random valid (keyboard/mouse) input, could crash all applications with random Win32 messages
- 2006: Crash only 7% of command-line utilities, but crashed 20/30 and hung 2/30 GUI applications on MacOS

SDU fuzz.html 15/34

Fuzz testing, today

Miller et al.'s early fuzzers were black-box: they had no domain knowledge of the application being tested.

Since then a new branch of white-box fuzzers have come forward:

- American Fuzzy Lop (AFL) named after a rabbit! http://lcamtuf.coredump.cx/afl/
- □ **libFuzzer** http://llvm.org/docs/LibFuzzer.html

Each of these fuzzers have an impressive trophy list of discovered bugs.



Coverage guided fuzzing

These fuzzers are coverage-guided, meaning you compile your program with special build tools.

The build tools instrument the compiled code so that the fuzzer

- can generate input which covers all possible paths
- can disregard input which tests paths already taken thereby avoiding tests of the same paths over and over

In addition they accept a corpus of typical test input files.

This will help point the fuzzer in the right direction, e.g., when the input format is tricky.

SDU 🎓

AFL, technically (1/2)

AFL's coverage guiding works along these lines:

```
queue = load user's test input;
while (queue not empty) {
   input = queue.next();
   input = trim(input);
   for (new_input in mutate(input)) {
      run program (new_input) with instrumentation;
      if (run visits new program path) {
         queue.add(new_input);
```

Input trimming (line 4) should cut down input size, but preserve the measured program behavior.

Instrumentation tells if a new program path was taken.

18/34

AFL, technically (2/2)

Underneath the hood AFL uses a number of mutation strategies to alter input:

- □ bitflips
- □ arithmetic
- □ known, "interesting" value overwrites
- a combined "havoc" strategy that combines bitflips, overwrites, block deletion, block duplication, ...

AFL, technically (2/2)

Underneath the hood AFL uses a number of mutation strategies to alter input:

- □ bitflips
- □ arithmetic
- □ known, "interesting" value overwrites
- a combined "havoc" strategy that combines bitflips, overwrites, block deletion, block duplication, ...

In the words of the author: "ultimately, it's never easy to get from Set-Cookie: FOO=BAR to Content-Length: -1 by randomly flipping bits."

For this reason AFL also lets the user supply a dictionary of domain-relevant tokens.

Example: Fuzzing ministat

In the past I've used the command line tool ministat

Given a text file in/chameleon of sample numbers

ministat will compute some basic statistics:

\$ X :	./mini in/cha	nistat in/chameleon chameleon							
X 		۱	x	x <u>M</u> A	х		x		
+ x	 N 5	 Min 150	 Max 930	 Median 500	Avg 540	Stddev 299.08193	+		

So, ministat seems like a nice candidate for fuzzing.

SDU 🎓

Installing and running AFL

First I installed AFL

Second, I downloaded the source code for ministat
from here: https://github.com/thorduri/ministat

For AFL to use ASan: export AFL_USE_ASAN=1
I compiled it with instrumentation: make CC=afl-clang
(this should create an executable called ministat)

Now I run AFL: afl-fuzz -i in -o out ./ministat @@ where @@ take the place of the input filename, in is an testcase directory, and out is a directory of AFL's findings.

For programs expecting its input on the command line: SDU A afl-fuzz -i in -o out program-path 21/34

The rabbit runs...

american fuzzy lo	op 2.52b (minista	t)		
process timing ————————————————————————————————————		— overall results ———		
run time : 0 days, 0 hrs, 1 mi	n, 13 sec	cycles done : 0		
last new path : 0 days, 0 hrs, 0 mi	n, 35 sec	total paths : 36		
last uniq crash : 0 days, 0 hrs, 0 mi	n, 54 sec	uniq crashes : 1		
last uniq hang : none seen yet		uniq hangs : 0		
— cycle progress —————————	— map coverage —			
now processing : 8 (22.22%)	map density	: 0.20% / 0.23%		
paths timed out : 0 (0.00%)	<pre>count coverage : 1.88 bits/tuple</pre>			
— stage progress ———————	— findings in de	pth		
now trying : havoc	favored paths : 7 (19.44%)			
stage execs : 315/1536 (20.51%)	new edges on : 9 (25.00%)			
total execs : 48.1k	total crashes : 1 (1 unique)			
exec speed : 747.6/sec	total tmouts : 5 (1 unique)			
— fuzzing strategy yields —————		– path geometry ————		
bit flips : 10/624, 2/621, 1/615		levels : 2		
byte flips : 0/78, 0/75, 0/69		pending : 34		
arithmetics : 1/4359, 0/986, 0/0		pend fav : 6		
known ints : 0/400, 0/2094, 1/3036		own finds : 34		
dictionary : 0/0, 0/0, 0/214		imported : n/a		
havoc : 20/34.3k, 0/0		stability : 100.00%		
trim : 53.01%/34, 0.00%	_			
		[cpu: 49%]		

AFL continues to run. You can stop it with Ctrl-C.

- After a successful run AFL has produced a number of directories:
- **queue**/ the queue of input data
- crashes/ saved input data causing a crash
- hangs/ saved input data causing a timeout
- I tried both with and without ASan:
 - run 1: 1 crash in 1 minute (w/o ASan) run 2: 5 crashes in 5 minutes (w/o ASan) run 3: 3 crashes in 3 minutes (w/ASan)

SDU 🎓

afl-tmin: a separate shrinking tool (1/3)

AFL comes with afl-tmin: a separate tool for test minimization (shrinking)

afl-tmin expects

- a problematic input file (as written by AFL)
- output file name (of shrunk output)
- name of the program under test

The tool takes multiple passes (each separated into stages) until it cannot cut an input any further down...



afl-tmin: a separate shrinking tool (2/3)

\$ afl-tmin -i out/crashes/id\:000000\,sig\:11\,src\:000000\,op\:havoc\,rep\:2 -o shrunk.txt ./ministat @@
afl-tmin 2.52b by <lcamtuf@google.com>

- [+] Read 29 bytes from 'out/crashes/id:000000,sig:11,src:000000,op:havoc,rep:2'.
- [*] Performing dry run (mem limit = 50 MB, timeout = 1000 ms)...
- [+] Program exits with a signal, minimizing in crash mode.
- [*] **Stage #0:** One-time block normalization...
- [+] Block normalization complete, 21 bytes replaced.
- [*] --- Pass #1 ---
- [*] Stage #1: Removing blocks of data... Block length = 1, remaining size = 29
- [+] Block removal complete, 19 bytes deleted.
- [*] **Stage #2:** Minimizing symbols (4 code points)...
- [+] Symbol minimization finished, 0 symbols (0 bytes) replaced.
- [*] **Stage #3:** Character minimization...
- [+] Character minimization done, 0 bytes replaced.
- [*] --- Pass #2 ---
- [*] **Stage #1:** Removing blocks of data...
 - Block length = 1, remaining size = 10
- [+] Block removal complete, 0 bytes deleted.

```
File size reduced by : 65.52% (to 10 bytes)
Characters simplified : 210.00%
Number of execs done : 52
Fruitless execs : path=26 crash=0 hang=0
```

- [*] Writing output to 'shrunk.txt'...
- [+] We're done here. Have a nice day!

```
$
```

SDU 🎓

afl-tmin: a separate shrinking tool (3/3)

So afl-tmin managed to cut this crashing input:

\$F 1E1500 400 720 500 930



afl-tmin: a separate shrinking tool (3/3)

So afl-tmin managed to cut this crashing input:

\$F
1E1500
400
720
500
930

Down to this:

0 1E1000 0

without any domain-specific knowledge of ministat...

```
Not too bad :-)
```

SDU 👉

Fuzzing in the Bigger Picture



Fuzz testing, continuously

Google has since created OSS-Fuzz: – an effort to continuously fuzz open source software

https://github.com/google/oss-fuzz/

Projects in OSS-Fuzz will automatically have the project fuzz tested, e.g., on source code changes.

Underneath the hood OSS-Fuzz uses a distributed version of libFuzzer, named ClusterFuzz.

When I last checked OSS-Fuzz listed over 5000 bugs attributed to this effort!

To encourage open source projects to incorporate fuzzing, Google rewards (read: \$\$\$) projects that do so:

https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html 28/34

Within traditional manual testing one typically distinguishes between:

White-box testing: the tester has access to the source code for the system under test

Black-box testing: the tester knows only the API (or specification) of the system under test

Within automatized testing (QuickCheck + fuzzing) this distinction is more blurred:

Who has access – machine or human?

A traditional distinction, revisited

With QuickCheck we can black-box test a system only based on the API (or specification)

With QuickCheck we can white-box test a system, e.g., letting the tester study the source code to formulate generators and properties.

With AFL one typically white-box tests a system, e.g., by letting AFL inspect how input triggers different paths in the code.



Fuzzing as property-based testing (1/2)

Michal Zalewski, the AFL author, also sees value in testing for properties more broadly:

11) Going beyond crashes

Fuzzing is a wonderful and underutilized technique for discovering non-crashing design and implementation errors, too. Quite a few interesting bugs have been found by modifying the target programs to call abort() when, say:

- Two bignum libraries produce different outputs when given the same fuzzer-generated input,
- An image library produces different outputs when asked to decode the same input image several times in a row,
- A serialization / deserialization library fails to produce stable outputs when iteratively serializing and deserializing fuzzer-supplied data,
- A compression library produces an output inconsistent with the input file when asked to compress and then decompress a particular blob.
- From http://lcamtuf.coredump.cx/afl/README.txt

I recognize interesting properties (some well-known)...

SDU 🎓

Fuzzing as property-based testing (2/2)

So, even though not designed for such, with little "driver programs" that abort on a failed property, AFL can be used for a form of property-based testing. With afl-tmin we even have a shrinker!

Pro: you don't have to write ad hoc generators Con: you have little control over the generator

It is interesting how the two approaches are converging

QuickCheck Still focused on positive testing How to generate invalid input in general, e.g., to find security issues?

Fuzzing Still focused on negative testing **SDU** w to, e.g., build models, e.g., to test AUTOSAR? 32/

Marrying QuickCheck and fuzz testing?

Crowbar for OCaml marries QuickCheck and fuzzing:

- Using a compiler patch it emits instr. code to realize white-box generators for property-based testing
- It shows impressive abilities on the module-level

https://github.com/stedolan/crowbar

Marrying QuickCheck and fuzz testing?

Crowbar for OCaml marries QuickCheck and fuzzing:

- Using a compiler patch it emits instr. code to realize white-box generators for property-based testing
- It shows impressive abilities on the module-level

https://github.com/stedolan/crowbar

Conventional (manual, human-in-the-loop) wisdom: White-box testing is only practical up to a certain point. Is this also true for automatized tests?

Marrying QuickCheck and fuzz testing?

Crowbar for OCaml marries QuickCheck and fuzzing:

- Using a compiler patch it emits instr. code to realize white-box generators for property-based testing
- It shows impressive abilities on the module-level

https://github.com/stedolan/crowbar

Conventional (manual, human-in-the-loop) wisdom: White-box testing is only practical up to a certain point. Is this also true for automatized tests?

Perhaps once "lower-level properties" have been established for modules individually with white-box generators, one should switch to black-box generators and higher-level properties for integration testing? Today we've looked at several things:

- □ the challenges of C programming, memory-wise
- ASan, MSan, and UBSan that can help address these challenges
- \Box fuzz testing,
 - historically,
 - presently (AFL, libFuzzer, ...),
 - technically (coverage guiding, ...)
- compared fuzzing with property-based testing