CSE509 Computer System Security

Stony Brook University

2023-02-21

Vulnerability Discovery

Michalis Polychronakis

Stony Brook University

Defending against (0day) Vulnerability Exploitation

Finding and fixing bugs *(this lecture)*

- Sanitizers, fuzzing, symbolic execution, bug bounties, …
- *Who* will find the next 0-day?

~~Retrofit memory safety to C/C++~~ ➔ *rewrite critical components in Rust/Go*

- Eradicate the root cause of the problem: *memory errors*
- Performance and compatibility challenges
- *No protection against transient execution attacks (!)* *(future lecture)*

Exploit mitigations *(next lecture)*

- Assuming a vulnerability exists, "raise the bar" for exploitation
- DEP, GS, SafeSEH, SEHOP, ASLR, CFI, sandboxing, attack surface reduction, …

Finding Bugs

Code auditing: code reviews, code formatting, reverse engineering, …

Vulnerability scanning: known vulnerabilities, attack patterns, …

Automated testing: unit testing, regression testing, …

Static analysis: taint tracking, symbolic execution, …

Sanitizers (dynamic analysis): memory corruption, race conditions, …

Fuzzing: "dumb" / "smart" fuzzing, black/grey/white-box fuzzing, …

…

Vulnerability Scanning

Information sources

- Global: NVD (National Vulnerability Database), proprietary feeds, …
- Per-distribution: Ubuntu, Red Hat, …

OVAL: Open Vulnerability and Assessment Language

- "A community-developed language for determining vulnerability and configuration issues on computer systems"

One-time vs. regular scanning

- Even if a server/container won't ever change, it still has to be scanned regularly
- Vulnerabilities in old code keep being discovered
- Example: Shellshock (command exec vulnerability in Bash) was found in 2014; the bug was in code written in 1989

Static Code Analysis

Analyze code (without running it) to identify bugs/vulnerabilities

- In contrast to dynamic analysis, which depends on program input

Various techniques (and combinations)

- Data flow analysis
- Control flow graph analysis
- Taint tracking
- Model checking
- Symbolic execution
- Abstract interpretation

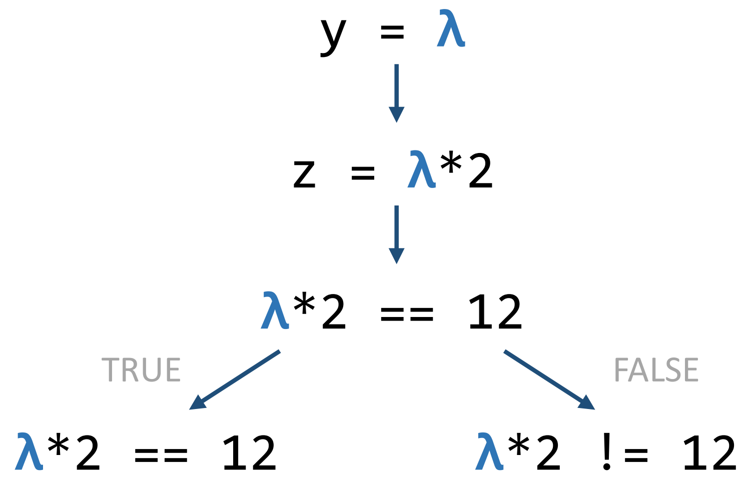
Vast number of free and commercial tools

Symbolic Execution

Goal: determine what inputs cause each part of a program to execute

- Symbolic values instead of actual data (abstract interpretation)
- Expressions for variables and statements in the program
- Constraints for the possible outcomes of conditional branches

```
int f() {
    y = read();
    z = y * 2;
    if (z == 12) {
        fail();
    } else {
        printf("OK");
    }
}
```


$$\begin{tikzcd}
& y = \lambda \arrow[d] & \\
& z = \lambda*2 \arrow[d] & \\
& \lambda*2 == 12 \arrow[dl, "\text{TRUE}"'] \arrow[dr, "\text{FALSE}"] & \\
\lambda*2 == 12 & & \lambda*2 \mathrel{!=} 12
\end{tikzcd}$$

Symbolic Execution

Main benefit: simultaneously explores multiple paths that a program may take under different inputs

- Each symbolic execution path represents the whole set of runs whose concrete values satisfy the path condition
- Uses a constraint solver to generate inputs that exercise a given path
- Very useful for "passing" difficult checks such as magic numbers or checksums

Main drawback: extremely slow even for moderately complex code

- Path/state explosion: number of paths increases exponentially with each branch (loops on symbolic variables are even worse)

Compromise: *concolic* execution (from "concrete + symbolic")

- Hybrid approach that performs symbolic execution along a concrete execution path

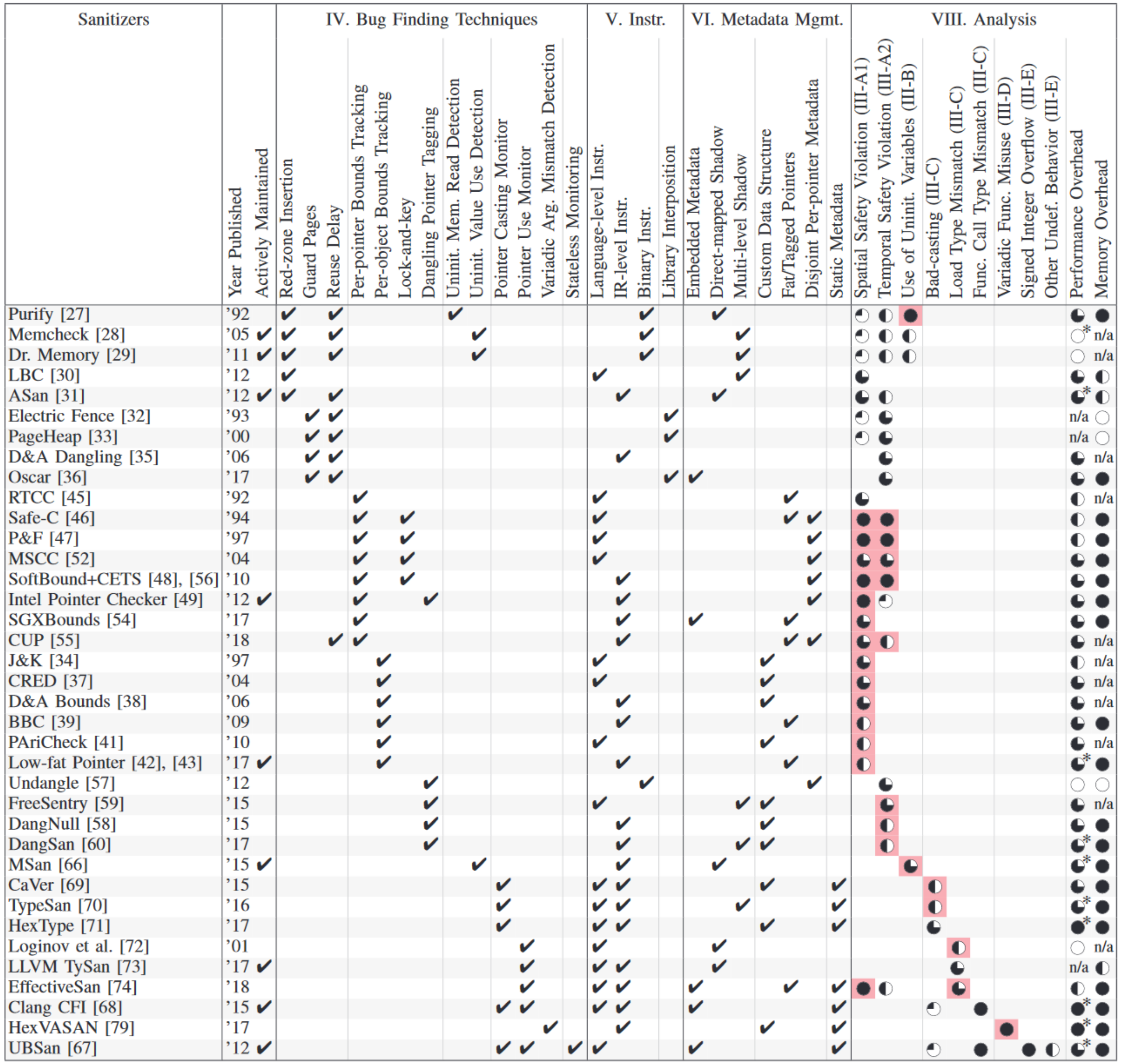
Sanitizers and Dynamic Testing

Instrument the program with various types of dynamic checks and detect errors at runtime

- Source code: compiler-level pass to add instrumentation
- Binary only: static or dynamic binary instrumentation (Pin, DynamoRio, Valgrind, …)

Various types of analysis

- Memory debugging
- Memory leak detection
- Race condition detection
- Profiling (execution, heap, cache, …)



Sanitizers			IV. Bug Finding Techniques													V. Instr.				VI. Metadata Mgmt.							VIII. Analysis										
	Year Published	Actively Maintained	Red-zone Insertion	Guard Pages	Reuse Delay	Per-pointer Bounds Tracking	Per-object Bounds Tracking	Lock-and-key	Dangling Pointer Tagging	Uninit. Mem. Read Detection	Uninit. Value Use Detection	Pointer Casting Monitor	Pointer Use Monitor	Variadic Arg. Mismatch Detection	Stateless Monitoring	Language-level Instr.	IR-level Instr.	Binary Instr.	Library Interposition	Embedded Metadata	Direct-mapped Shadow	Multi-level Shadow	Custom Data Structure	Fat/Tagged Pointers	Disjoint Per-pointer Metadata	Static Metadata	Spatial Safety Violation (III-A1)	Temporal Safety Violation (III-A2)	Use of Uninit. Variables (III-B)	Bad-casting (III-C)	Load Type Mismatch (III-C)	Func. Call Type Mismatch (III-C)	Variadic Func. Misuse (III-D)	Signed Integer Overflow (III-E)	Other Undef. Behavior (III-E)	Performance Overhead	Memory Overhead
Purify [27]	’92		✓		✓					✓								✓			✓						◔	◑	●							◕	●
Memcheck [28]	’05	✓	✓		✓						✓							✓				✓					◔	◑	◑							○*	n/a
Dr. Memory [29]	’11	✓	✓		✓						✓							✓				✓					◔	◑	◑							○	n/a
LBC [30]	’12		✓													✓						✓					◕									◕	◑
ASan [31]	’12	✓	✓		✓												✓				✓						◕	◑								◕*	◑
Electric Fence [32]	’93			✓	✓														✓								◔	◕								n/a	○
PageHeap [33]	’00			✓	✓														✓								◔	◕								n/a	○
D&A Dangling [35]	’06			✓	✓												✓											◕								◕	n/a
Oscar [36]	’17			✓	✓														✓	✓								◕								◕	●
RTCC [45]	’92					✓										✓								✓			◕									◑	n/a
Safe-C [46]	’94					✓		✓								✓								✓	✓		●	●								◑	●
P&F [47]	’97					✓		✓								✓									✓		●	●								◑	●
MSCC [52]	’04					✓		✓								✓									✓		◕	◕								◕	●
SoftBound+CETS [48], [56]	’10					✓		✓									✓								✓		●	●								◕	●
Intel Pointer Checker [49]	’12	✓				✓			✓								✓								✓		●	◔								◕	●
SGXBounds [54]	’17					✓											✓			✓				✓			◕									◕	●
CUP [55]	’18				✓	✓											✓							✓	✓		◕	◑								◕	n/a
J&K [34]	’97						✓									✓							✓				◕									◑	n/a
CRED [37]	’04						✓									✓							✓				◕									◕	n/a
D&A Bounds [38]	’06						✓										✓						✓				◕									◕	n/a
BBC [39]	’09						✓										✓							✓			◑									◕	●
PAriCheck [41]	’10						✓									✓							✓				◑									◕	n/a
Low-fat Pointer [42], [43]	’17	✓					✓										✓							✓			◑									◕*	●
Undangle [57]	’12								✓									✓							✓			◕								○	○
FreeSentry [59]	’15								✓							✓						✓	✓					◕								◕	n/a
DangNull [58]	’15								✓								✓						✓					◑								◕	●
DangSan [60]	’17								✓								✓					✓	✓					◑								◕*	●
MSan [66]	’15	✓									✓						✓				✓								◕							◕*	●
CaVer [69]	’15											✓				✓	✓						✓			✓				◑						◕	●
TypeSan [70]	’16											✓				✓	✓					✓				✓				◑						◕*	●
HexType [71]	’17											✓				✓	✓						✓			✓				◕						●*	●
Loginov et al. [72]	’01												✓			✓					✓										◑					○	n/a
LLVM TySan [73]	’17	✓											✓			✓	✓				✓										◕					n/a	◑
EffectiveSan [74]	’18												✓			✓	✓			✓				✓		✓	●	◑			◕					◑	●
Clang CFI [68]	’15	✓										✓	✓			✓	✓			✓						✓				◔		●				●*	●
HexVASAN [79]	’17													✓			✓						✓			✓							●			●*	●
UBSan [67]	’12	✓										✓	✓		✓	✓				✓						✓				◔		●		●	◑	◕*	●

Valgrind

Dynamic binary instrumentation framework
for building dynamic analysis tools

- Machine code is first translated into Valgrind’s intermediate representation (IR): processor-neutral, SSA-based form
- Valgrind tools perform transformations at the IR level
- Transformed IR is translated back into machine code and then the processor runs it (VM using JIT compilation)

No source code is needed!

- But DBI is very costly: orders of magnitude slower program execution

Several tools for memory error detection and profiling

Valgrind Tools

Memcheck

- Instruments all memory reads, writes, and calls to malloc/new/free/delete
- Shadow memory for every bit of data: tracks whether a bit is *defined* or not
- Can detect illegal memory accesses, use of uninitialized memory, memory leaks, double/mismatched free(), …

Cachegrind: cache profiler

Callgrind: call graph analyzer

Massif: heap profiler

Helgrind: race condition detector

…

Compiler-level Sanitizers (LLVM, GCC)

Specialized sanitizers based on compiler instrumentation

Main benefit: much faster

Main drawback: need for source code

ASan (Address Sanitizer)

Fast memory error detector: finds use-after-free and {heap, stack, global} buffer overflow bugs in C/C++ programs (2x slowdown)

TSan (Thread Sanitizer)

Data race detector (5-15x slowdown)

MSan (Memory Sanitizer)

Detects uses of uninitialized memory (3x slowdown)

UBSan (Undefined Behavior Sanitizer)

Fast undefined behavior detector

AddressSanitizer

Two parts: compiler instrumentation module and runtime library

- Directly-mapped shadow memory

Can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack, and globals
- Use after free
- Use after return
- Use after scope
- Double-free, invalid free
- Initialization order bugs
- Memory leaks (experimental)

Clang (3.1+), GCC (4.8+), and Xcode (7.0+)

ASan Example: Stack Buffer Overflow

```c
int main(int argc, char **argv) {
  int stack_array[100];
  stack_array[1] = 0;
  return stack_array[argc + 100];  // BOOM
}
```

```
# clang -O -g -fsanitize=address %t && ./a.out
==7405==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff64740634 at pc 0x46c103 bp
0x7fff64740470 sp 0x7fff64740468
READ of size 4 at 0x7fff64740634 thread T0
    #0 0x46c102 in main /tmp/example_StackOutOfBounds.cc:5

Address 0x7fff64740634 is located in stack of thread T0 at offset 436 in frame
    #0 0x46bfaf in main /tmp/example_StackOutOfBounds.cc:2

  This frame has 1 object(s):
    [32, 432) 'stack_array' <== Memory access at offset 436 overflows this variable
```

ASan Example: Golbal Buffer Overflow

```
int global_array[100] = {-1};
int main(int argc, char **argv) {
  return global_array[argc + 100];  // BOOM
}
```

```
# clang -O -g -fsanitize=address %t && ./a.out
==7455==ERROR: AddressSanitizer: global-buffer-overflow on address 0x000000689b54 at pc 0x46bfd8 bp
0x7fff515e5ba0 sp 0x7fff515e5b98
READ of size 4 at 0x000000689b54 thread T0
    #0 0x46bfd7 in main /tmp/example_GlobalOutOfBounds.cc:4

0x000000689b54 is located 4 bytes to the right of
  global variable 'global_array' from 'example_GlobalOutOfBounds.cc' (0x6899c0) of size 400
```

ASan Example: Heap Buffer Overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];  // BOOM
  delete [] array;
  return res;
}
```

```
# clang++ -O -g -fsanitize=address %t && ./a.out
==25372==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61400000ffd4 at pc 0x0000004ddb59 bp
0x7fffea6005a0 sp 0x7fffea600598
READ of size 4 at 0x61400000ffd4 thread T0
    #0 0x46bfee in main /tmp/main.cpp:4:13

0x61400000ffd4 is located 4 bytes to the right of 400-byte region [0x61400000fe40,0x61400000ffd0)
allocated by thread T0 here:
    #0 0x4536e1 in operator delete[](void*)
    #1 0x46bfb9 in main /tmp/main.cpp:2:16
```

ASan Example: Heap Use After Free

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

```
# g++ -O -g -fsanitize=address heap-use-after-free.cc && ./a.out
==5587==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe44 at pc 0x47b55f bp
0x7ffc36b28200 sp 0x7ffc36b281f8
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x47b55e in main /home/test/example_UseAfterFree.cc:7
    #1 0x7f15cfe71b14 in __libc_start_main (/lib64/libc.so.6+0x21b14)
    #2 0x47b44c in _start (/root/a.out+0x47b44c)

0x61400000fe44 is located 4 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x465da9 in operator delete[](void*) (/root/a.out+0x465da9)
    #1 0x47b529 in main /home/test/example_UseAfterFree.cc:6

previously allocated by thread T0 here:
    #0 0x465aa9 in operator new[](unsigned long) (/root/a.out+0x465aa9)
    #1 0x47b51e in main /home/test/example_UseAfterFree.cc:5
```

GWP-ASan: Sampling heap mem

chromium.org/Home/chromium-security/articles/gwp-asan

Incognito

The Chromium Projects

Search this site

Chromium > Chromium Security > Articles >

GWP-ASan: Sampling heap memory error detection in-the-wild

By Vlad Tsyrklevich, Dynamic Tools Teams

Memory safety errors, like use-after-frees and out-of-bounds reads/writes, are a leading source of vulnerabilities in C/C++ applications. Despite investments in preventing and detecting these errors in Chrome, over 60% of high severity vulnerabilities in Chrome are memory safety errors. Some memory safety errors don't lead to security vulnerabilities but simply cause crashes and instability.

Chrome uses state-of-the-art techniques to prevent these errors, including:

- Coverage-guided fuzzing with AddressSanitizer (ASan)
- Unit and integration testing with ASan
- Defensive programming, like custom libraries to perform safe math or provide bounds checked containers
- Mandatory code review

Chrome also makes use of sandboxing and exploit mitigations to complicate exploitation of memory errors that go undetected by the methods above.

AddressSanitizer is a compiler instrumentation that finds memory errors occurring on the heap, stack, or in globals. ASan is highly effective and one of the lowest overhead instrumentations available that detects the errors that it does; however, it still incurs an average 2-3x performance and memory overhead. This makes it suitable for use with unit tests or fuzzing, but not deployment to end users. Chrome used to deploy SyzyASAN instrumented binaries to detect memory errors. SyzyASAN had a similar overhead so it was only deployed to a small subset of users on the canary channel. It was discontinued after the Windows toolchain switched to LLVM.

GWP-ASan, also known by its recursive backronym, GWP-ASan Will Provide Allocation Sanity, is a sampling allocation tool designed to detect heap memory errors occurring in production with negligible overhead. Because of its negligible overhead we can deploy GWP-ASan to the entire

Sanitizers are not enough

Chromium and Firefox developers are active users of AddressSanitizer

- It has found hundreds of bugs in these web browsers
- Bugs also found in FFmpeg, FreeType, Safari, iTunes, Opera, …
- The Linux kernel has enabled ASan for x86-64 as of v4.0

They are still best-effort tools

- Only as good as the test inputs used

Not all types of bugs are covered

- Example: adjacent buffers in structs and classes are not protected from overflow (backwards compatibility problems)

Fuzzing

Automated software testing approach for identifying bugs ➔ potential vulnerabilities

- Feed arbitrary inputs to programs to find erroneous cases
- No or very little knowledge of the internal operations of the system under test (SUT)

A form of negative testing *(fault injection)*

- Unexpected, malformed, or semi-valid inputs instead of the properly formatted data expected by the program

Fuzzing has gradually evolved from a niche testing technique to a full software testing and vulnerability discovery discipline

- Related testing techniques: protocol mutation, robustness testing, syntax testing, dirty testing, …

History of Fuzzing

1983: The Monkey

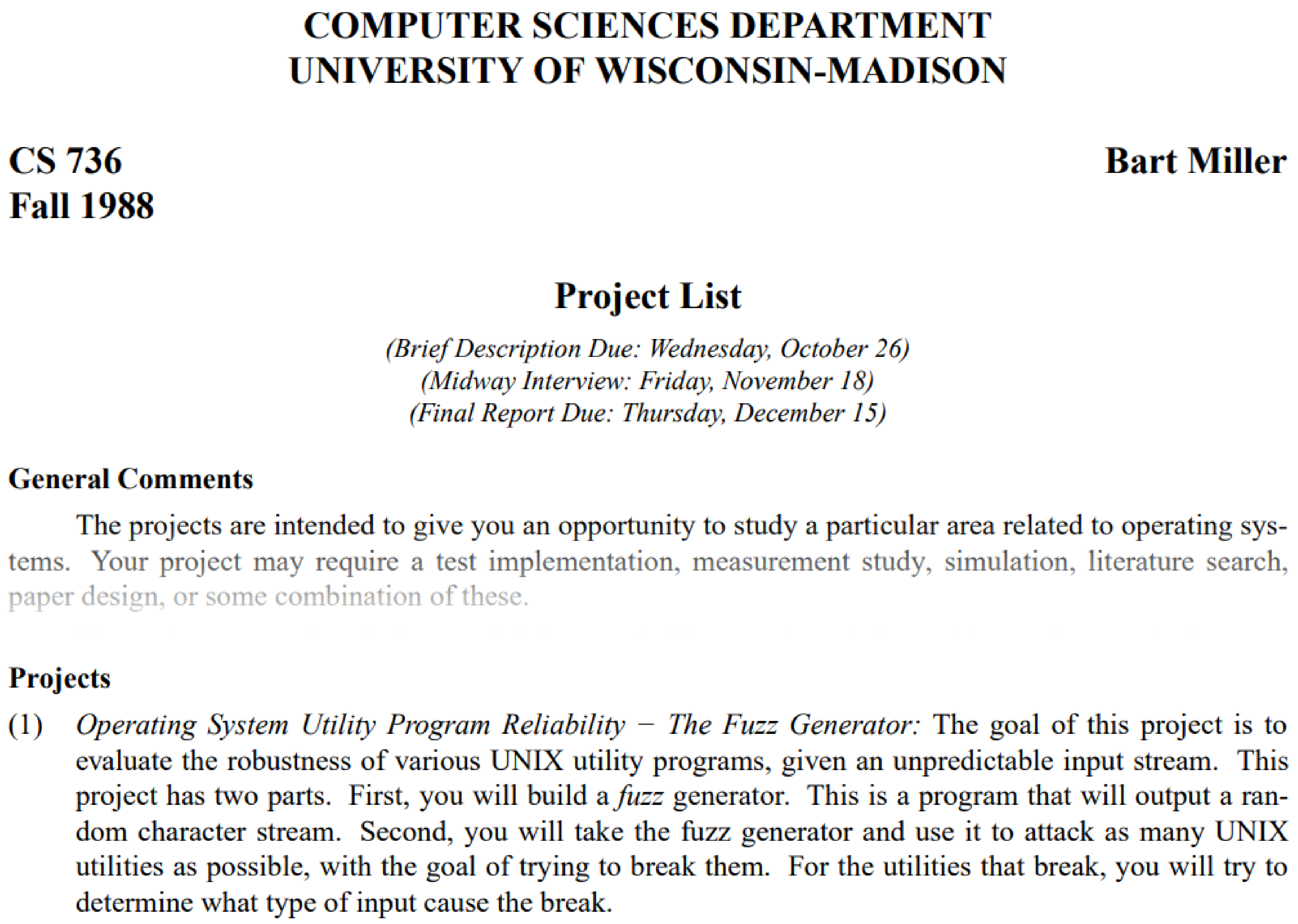
> *"…was a small desk accessory that used the journaling hooks to feed random events to the current application, so the Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon."*

1988: Bart Miller's fuzz project assignment

1990: Boris Beizer explains Syntax Testing

1990: "An Empirical Study of the Reliability of UNIX Utilities" [Miller et al.]

…



COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

CS 736
Fall 1988

Bart Miller

Project List

(Brief Description Due: Wednesday, October 26)
(Midway Interview: Friday, November 18)
(Final Report Due: Thursday, December 15)

General Comments

The projects are intended to give you an opportunity to study a particular area related to operating systems. Your project may require a test implementation, measurement study, simulation, literature search, paper design, or some combination of these.

Projects

(1) *Operating System Utility Program Reliability − The Fuzz Generator:* The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. This project has two parts. First, you will build a *fuzz* generator. This is a program that will output a random character stream. Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them. For the utilities that break, you will try to determine what type of input cause the break.

An Empirical Study of the Reliability

of

UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu

Summary

Operating system facilities, such as the kernel and utility programs, are typically assumed to be reliable. In our recent experiments, we have been able to crash 25-33% of the utility programs on any version of UNIX that was tested. This report describes these tests and an analysis of the program bugs that caused the crashes.

Fuzzing Like It's 1989

Trail of Bits conducted an interesting experiment

"… let's take a long look back 30 years and reflect on the original fuzzing paper, An Empirical Study of the Reliability of UNIX Utilities, and its 1995 follow-up, Fuzz Revisited, by Barton P. Miller. In this blog post, we are going to find bugs in modern versions of Ubuntu Linux using the exact same tools as described in the original fuzzing papers."

	Ubuntu 18.10 (2018)	**Ubuntu 18.04 (2018)**	**Ubuntu 16.04 (2016)**	**Ubuntu 14.04 (2014)**	**Slackware 2.1.0 (1995)**
Crashes	1 (f77)	1 (f77)	2 (f77, ul)	2 (swipl, f77)	4 (ul, flex, indent, gdb)
Hangs	1 (spell)	1 (spell)	1 (spell)	2 (spell, units)	1 (ctags)
Total Tested	81	81	81	81	55
Crash/Hang %	2%	2%	4%	5%	9%

Regression Testing vs. Fuzzing

Regression testing

Run program using well-defined normal (and abnormal) inputs and look for errors (typically manually selected)

Goal: prevent code changes or new code from introducing bugs

Fuzzing

Run program using a vast number of random (abnormal) inputs and look for errors

Goal: identify (potentially exploitable) bugs

Types of Fuzzing

Can be categorized across various dimensions

Input or attack vector

Network, API/protocol, GUI, files/media, …

Structure

Black, grey, or white box

Test case complexity

“Dumb” fuzzing: random inputs or simple templates

Generation-based: generate inputs from scratch based on existing grammars/models or simulation

Mutation-based: dynamic generation or evolution of new inputs based on a small set of real “seed” inputs

Generic Architecture

Three main components

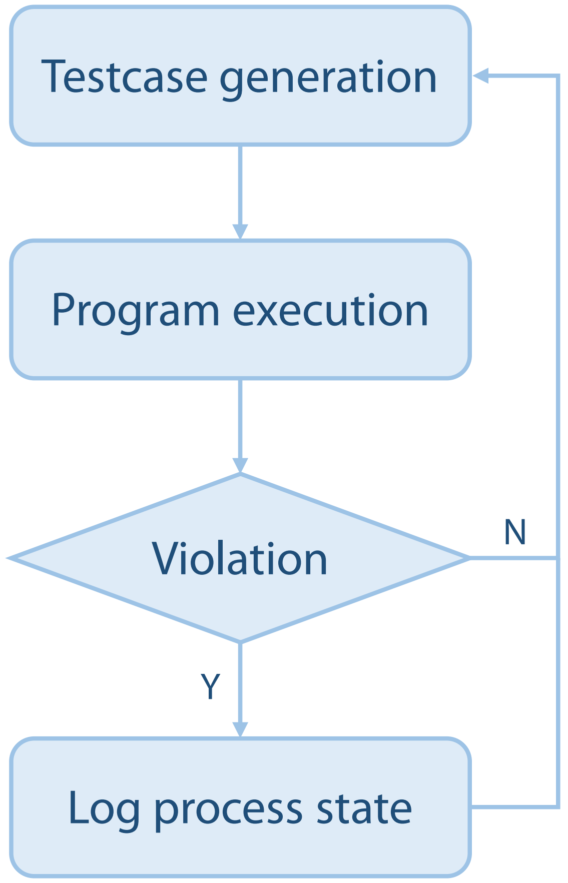
- Test case generator
- Execution engine
- Data logger/Assessor

An orchestrator manages the whole process

- Need for speed: fork servers, OS-assisted fuzzing, in-process fuzzing (avoid forking), …

Detect erroneous conditions beyond crashing

- Sanitizers can be integrated to detect security violations



Mutation-based Fuzzing

Start with a set of "seed" inputs for a given application and mutate them to generate new inputs

- libpng ➔ PNG images, Acrobat Reader ➔ PDF files, …

Various transformations: randomize fields, bit flipping, add/remove/modify data, …

Much simpler to implement, although some intelligence is needed

- Avoid useless test cases (e.g., re-compute checksums)
- Automated seed selection (or test suite reduction): pick the best seeds/part of input that is actually inducing the failure

Tools: *AFL, VUzzer, ZZUF, Taof, GPF, ProxyFuzz, Radamsa, …*

Generation-based Fuzzing

“Smart” or context-aware fuzzing: generate inputs from scratch based on file/protocol format specifications (RFC, documentation, …)

Data modelling, state modelling, …

- Extra knowledge ➔ higher quality test cases
- Get “deeper” into protocol/application states
- Generate more valid test cases that are not immediately rejected

Mutations are still applied, but with finer granularity

- Per message, per field, per object, …

Main drawback: accurate modeling requires significant manual effort

Tools: *SPIKE, Sulley, Mu-4000, Codenomicon, Peach Fuzzer, …*

Black-box Fuzzers (“Traditional” Fuzzing)

They treat the program as a black box and are unaware of internal program structure

- Only the input and output of the SUT are known
- Randomized inputs or simple heuristic/grammar-based test case generation
- No feedback: each test is independent from the rest

Main benefits: simple and fast, can be easily parallelized, can work with very complex programs

Main drawback: can expose only “shallow” bugs

Tools: *Peach, Protos, Spike, Autodafe, ansvif, …*

White-box Fuzzers

Use heavyweight program analysis to

- Understand the impact of inputs (beyond crashing)
- Increase code coverage and reach critical program locations

Take advantage of program specification (if available) to generate (sequences of) inputs

- E.g., model-based testing: program output can also be checked against the specification
- E.g., symbolic execution: collect constraints on inputs, negate those, solve with constraint solver, generate new inputs

Main benefit: can expose "deep" bugs

Main drawbacks: very slow, need source code

Tools: *KLEE, S2E, DART, SAGE (Microsoft), Dowser, …*

Grey-box Fuzzers

Rely on code instrumentation to get information about the program and inform subsequent tests

- E.g., monitor program states reached under different inputs, and guide subsequent inputs to increase code coverage
- E.g., use data taint tracking to identify which registers and memory locations are related to which part of the input

Code coverage approaches:

- Statement coverage: which statements have been executed
- Branch coverage: which branches have been taken
- Path coverage: which paths were taken.

Tools: *AFL, VUzzer, Driller, honggfuzz, libFuzzer, …*

AFL (American Fuzzy Lop)

Written by Michał Zalewski

Practical, open-source, security-oriented fuzzer

- Has had a huge impact in the security community

Uses compilation-time instrumentation and genetic algorithms to discover clean, interesting test cases

- Goal: trigger new internal states in the targeted binary
- Black-box testing is supported too if source code is not available

Evolutionary fuzzing: feedback loop to assess how good an input is

- AFL retains any input that discovers a new path and mutates that input further to check if doing so leads to new basic blocks

AFL Usage

User provides a sample command that runs the tested application and at least one small example input file

- E.g., in case of an audio player, AFL can be instructed to open a short sound file with it

AFL attempts to execute the specified command

- If that succeeds, it tries to reduce the input file to the smallest one that triggers the same behavior

AFL then begins the actual fuzzing by applying various modifications to the input file

- Inputs that result in crashes or hangs are saved for further user inspection

AFL Mutation Strategies

Bit or byte flips

Simple arithmetic on byte values

Known integers (e.g., -1, 256, 1024, MAX_INT-1, MAX_INT)

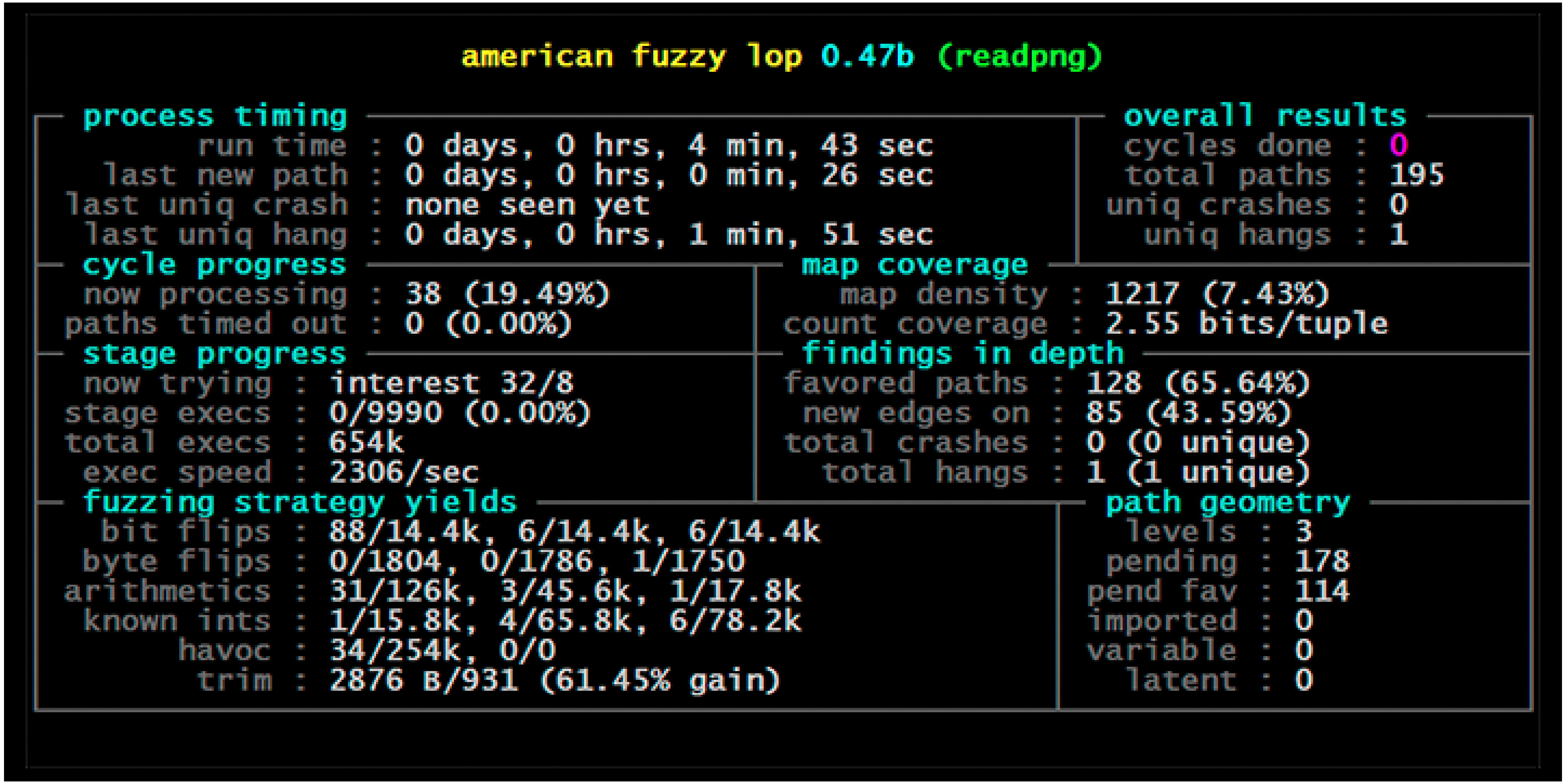
Attempt to set "interesting" bytes, words, or dwords

Addition or subtraction of small integers to bytes, words, or dwords

Completely random single-byte sets

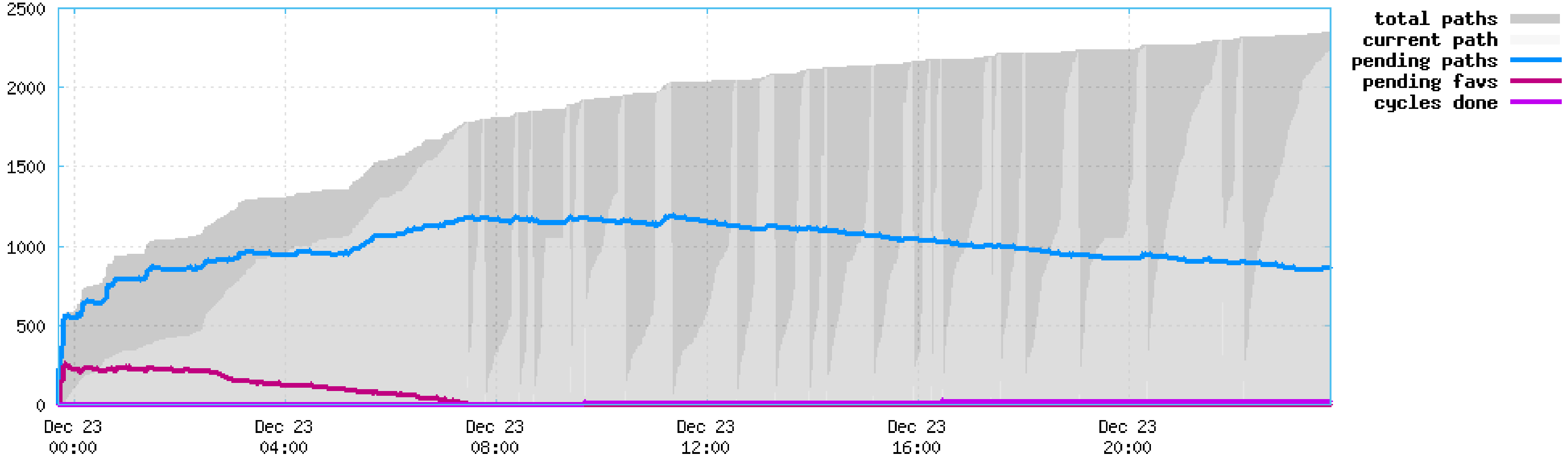
Block delete, duplicate via overwrite/insertion, or memset

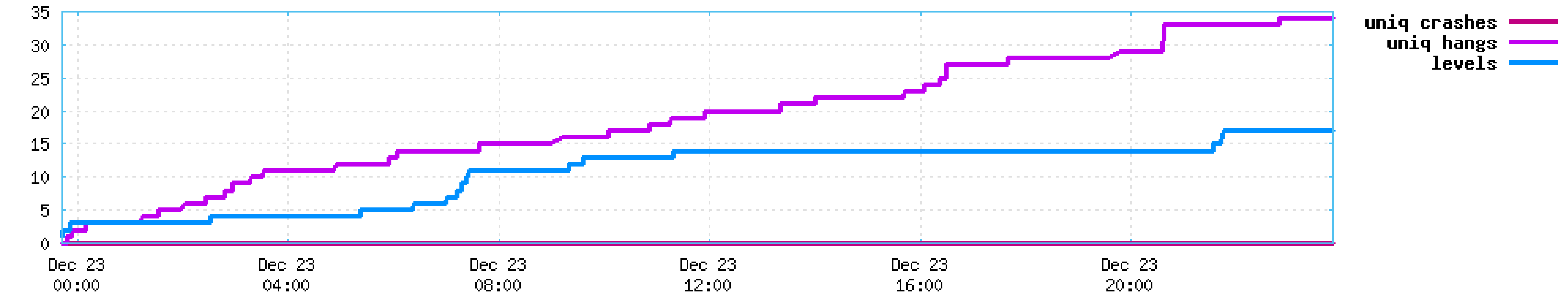
Test case splicing: take two distinct files that differ in at least two locations and splice them at a random location

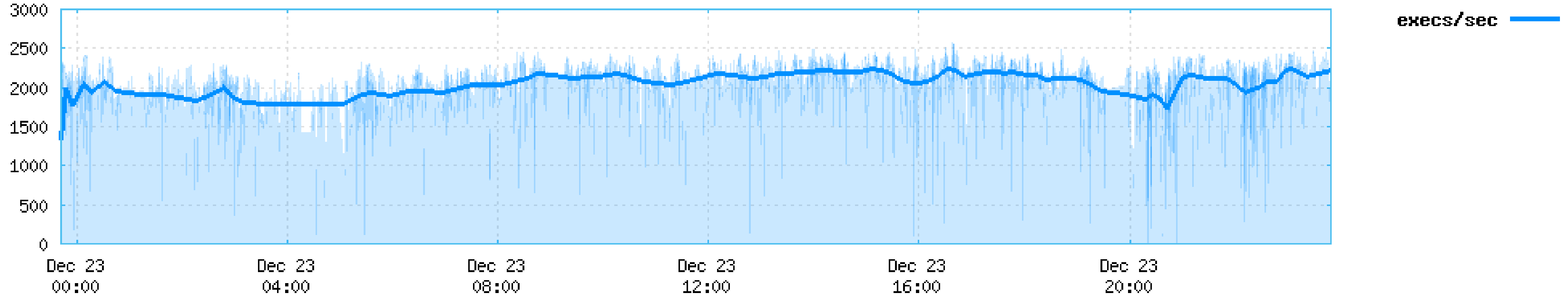


```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1      │
├─ cycle progress ───────────────────┬─ map coverage ──┴───────────────────────┤
│  now processing : 38 (19.49%)      │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)        │ count coverage : 2.55 bits/tuple       │
├─ stage progress ───────────────────┼─ findings in depth ─────────────────────┤
│  now trying : interest 32/8        │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)       │  new edges on : 85 (43.59%)            │
│ total execs : 654k                 │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec             │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ──────────┴─────────────┬─ path geometry ───────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k         │    levels : 3             │
│  byte flips : 0/1804, 0/1786, 1/1750             │   pending : 178           │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k          │  pend fav : 114           │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k          │  imported : 0             │
│       havoc : 34/254k, 0/0                       │  variable : 0             │
│        trim : 2876 B/931 (61.45% gain)           │    latent : 0             │
└──────────────────────────────────────────────────┴───────────────────────────┘
```


2500
2000
1500
1000
500
0
Dec 23 00:00
Dec 23 04:00
Dec 23 08:00
Dec 23 12:00
Dec 23 16:00
Dec 23 20:00
total paths
current path
pending paths
pending favs
cycles done


35
30
25
20
15
10
5
0
Dec 23 00:00
Dec 23 04:00
Dec 23 08:00
Dec 23 12:00
Dec 23 16:00
Dec 23 20:00
uniq crashes
uniq hangs
levels


3000
2500
2000
1500
1000
500
0
Dec 23 00:00
Dec 23 04:00
Dec 23 08:00
Dec 23 12:00
Dec 23 16:00
Dec 23 20:00
execs/sec

LibFuzzer (part of LLVM)

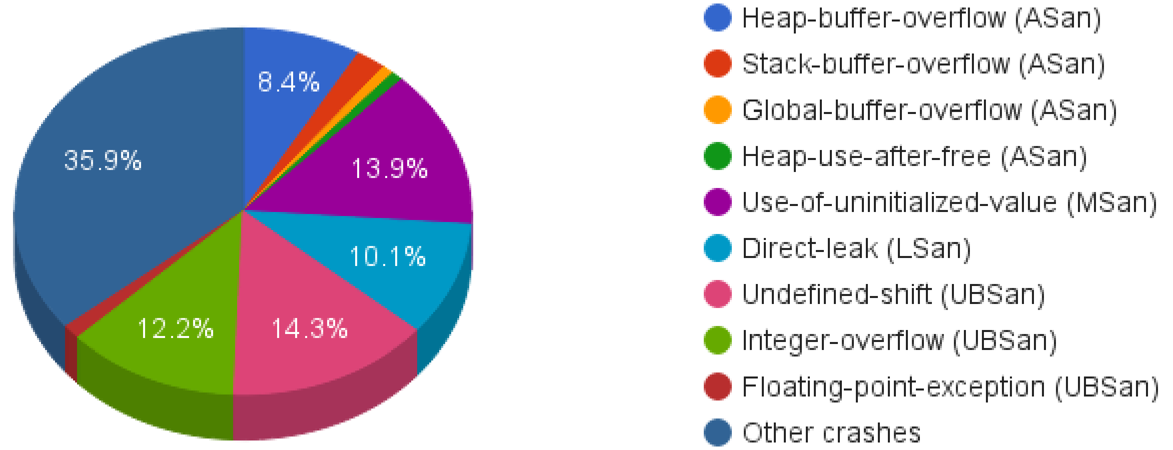
In-process, coverage-guided, evolutionary fuzzing engine

- In-process: inputs are mutated directly in memory (no need to launch a new process for every test case)
- Coverage-guided: prioritize test cases that increase overall coverage
- White-box: compilation-time instrumentation of the source code

Can fuzz individual libraries

Support for sanitizers

- ASan
- UBSan
- MSan



OSS-Fuzz: Continuous Fuzzing for Open Source Software

Free fuzzing platform for open-source projects

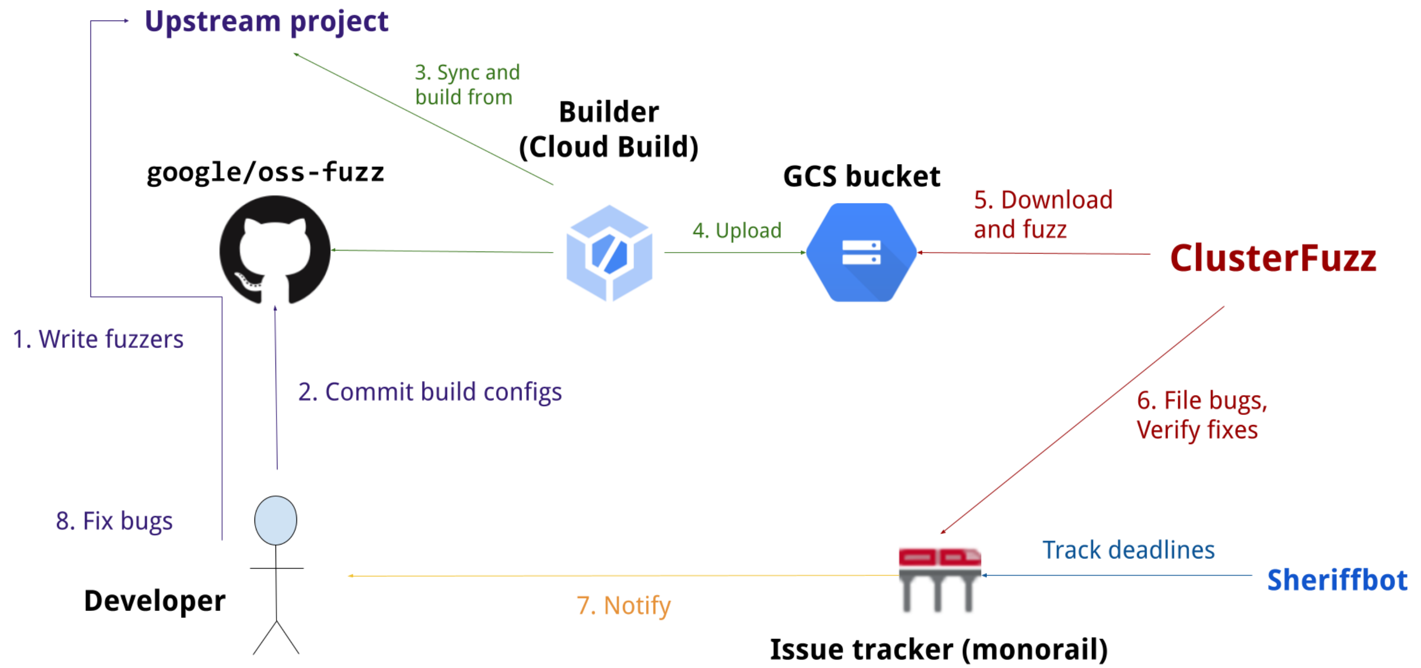
- Since 2016: over 8,800 vulnerabilities and 28,000 bugs fixed across 850 projects
- Awarded over $600,000 to contributors for integrating new projects into OSS-Fuzz

Fuzzing engines

- libFuzzer, AFL++, Honggfuzz

ClusterFuzz

- Distributed fuzzer execution environment and reporting



Determining Exploitability

Crash != vulnerability

Look for cases where data is written to controllable addresses or registers ➔ ability to control code execution

- Are saved return address/stack variables overwritten?
- Is the crash in a heap management function?
- Are registers values derived from data sent by the fuzzer?
- Is the crash triggered by a read operation?
- Can we craft a test case to avoid this?
- Is the crash triggered by a write operation?
- Do we have full or partial control of the faulting address?
- Do we have full or partial control of the written value?

Use case: Microsoft's !Exploitable

Windows debugging extension (Windbg)

Provides automated crash analysis and security risk assessment

- Assigns an exploitability rating to crashes
- Uses lightweight taint tracking

Use stack snapshot (hash) as a "fingerprint" for a crash

- Uses a set of patterns to filter out stack frames which are used in processing exceptions or are OS resource functions
- Ignore uninteresting parts of the stack ➔ more relevant hashes
- Grouping crashes dramatically reduces the number of crashes that need to be manually inspected

DARPA Cyber Grand Challenge (CGC)

CTF-like contest to develop *autonomous* systems that can discover, exploit, and patch vulnerabilities

- Multiple rounds, final event on August 4, 2016 (DEFCON 24)
- 32-bit challenge binaries (vastly simplified ABI)

Exploitation demonstrated by submitting a *"proof of vulnerability"*

- A binary program that communicates with the opponent's binary and exploits it

Bug finding

- Most teams used a combination of AFL-like fuzzing and symbolic execution
- Collectively, all teams found vulnerabilities in 99 out of the 131 challenge binaries

CGC Use Case: Mechanical Phish (UCSB, 3rd place)

Fully open source, built from open-source components

Driller: bug finding component

- Fuzzing (AFL) + symbolic execution
- QEMU for execution monitoring

Adding symbolic execution

- Path explosion ➔ used only selectively
- Symbolic execution only for paths that AFL finds interesting
- Identify transitions that have not been seen yet, and feed the corresponding inputs back to AFL

```
----------------------------------------
|               Driller                |
|                                      |
| ++++++++++++++        ++++++++++++++ |
| + angr       + =====> + AFL        + |
| +            +        +            + |
| + Symbolic   +        + Genetic    + |
| + Tracing    + <===== + Fuzzing    + |
| ++++++++++++++        ++++++++++++++ |
|                                      |
----------------------------------------
```