



# Fuzzing: The Age of Vulnerability Discovery

Richard Johnson  
Fuzzing IO

Workshop on Offensive  
Technologies 2023

# Agenda

Introduction

The Age of Vulnerability Discovery

Advanced Instrumentation

Improved Input Generation

Reaching New Attack Surface

Conclusions





## Richard Johnson

Owner, Fuzzing IO

Advanced Fuzzing and Crash Analysis Training  
Contract fuzzing harness and security tool development

Contact

[rjohnson@fuzzing.io](mailto:rjohnson@fuzzing.io)

[@richinseattle](https://twitter.com/richinseattle)

whoami



FUZZING/IO



# Introduction

- In 2020, I discussed fuzzing in "Lightning in a Bottle" reviewing major milestones that led to "The Fuzzing Renaissance" that was kicked off by the creation of effective greybox mutational parser fuzzing with the release of American Fuzzy Lop.



# Introduction

- The Renaissance was a historic period of cultural, artistic, political and economic “rebirth” hat attempted to surpass ideas and achievements of antiquity. In many ways this is a great metaphor for the rebirth of interest and expanded capabilities of fuzzing starting in 2013.



# Introduction

- Today I'd like to reflect on this impact and show how we, as a community at the intersection of academic and professional research, have participated in this so-called Renaissance which has now led to a new age – The Age of Vulnerability Discovery.

# Remembering How We Got Here

AMERICAN FUZZY LOP - ZALEWSKI, 2013

- Edge transitions are encoded as tuple and tracked in global map
- Includes coverage and frequency
- Simplified genetic algorithm for continual improvement of input generation
- Uses dictionaries and traditional mutation fuzzing strategies

```

american fuzzy lop 2.52b (dump-prog)

process timing | overall results
  run time : 0 days, 0 hrs, 2 min, 7 sec | cycles done : 0
  last new path : 0 days, 0 hrs, 0 min, 48 sec | total paths : 634
  last uniq crash : none seen yet | uniq crashes : 0
  last uniq hang : none seen yet | uniq hangs : 0
cycle progress | map coverage
  now processing : 15 (2.37%) | map density : 0.39% / 3.33%
  paths timed out : 0 (0.00%) | count coverage : 4.55 bits/tuple
stage progress | findings in depth
  now trying : bitflip 2/1 | favored paths : 227 (35.80%)
  stage execs : 2054/3119 (65.85%) | new edges on : 256 (40.38%)
  total execs : 155k | total crashes : 0 (0 unique)
  exec speed : 1331/sec | total tmouts : 1 (1 unique)
fuzzing strategy yields | path geometry
  bit flips : 0/22.5k, 0/19.3k, 0/19.3k | levels : 2
  byte flips : 0/2417, 0/395, 0/431 | pending : 627
  arithmetics : 2/21.1k, 0/21.8k, 0/21.5k | pend fav : 221
  known ints : 0/1081, 0/4491, 0/8749 | own finds : 3
  dictionary : 0/0, 0/0, 0/428 | imported : n/a
  havoc : 1/3828, 0/0 | stability : 100.00%
  trim : 0.00%/1287, 83.88%

```

[cpu000: 37%]

# The Age of Vulnerability Discovery



**As of February 2023, ClusterFuzz has found ~27,000 bugs in Google and over 8,900 vulnerabilities and 28,000 bugs across 850 projects integrated with OSS-Fuzz.**

- ClusterFuzz README



# “Evaluating Fuzz Testing”

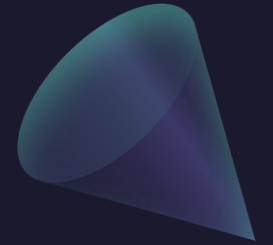
GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS - 2018

- “We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every **evaluation** we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments.”

# “Evaluating Fuzz Testing”

GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS - 2018

- Until 2018, paper evaluations were ad-hoc and suffered from errors
  - Most papers fail to perform multiple runs and used short timeouts
  - Most papers counted crashes using noisy AFL measurements instead of bugs and coverage
  - Most papers did not consider impact of initial seed selection
  - Most papers vary the target set too widely and may have reported selective results
    - COREUTILS / SPEC2000, SYNTHETIC TEST SUITES (CGC), LAVA

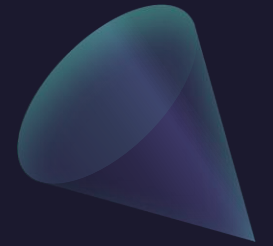


# “Evaluating Fuzz Testing”

GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS - 2018

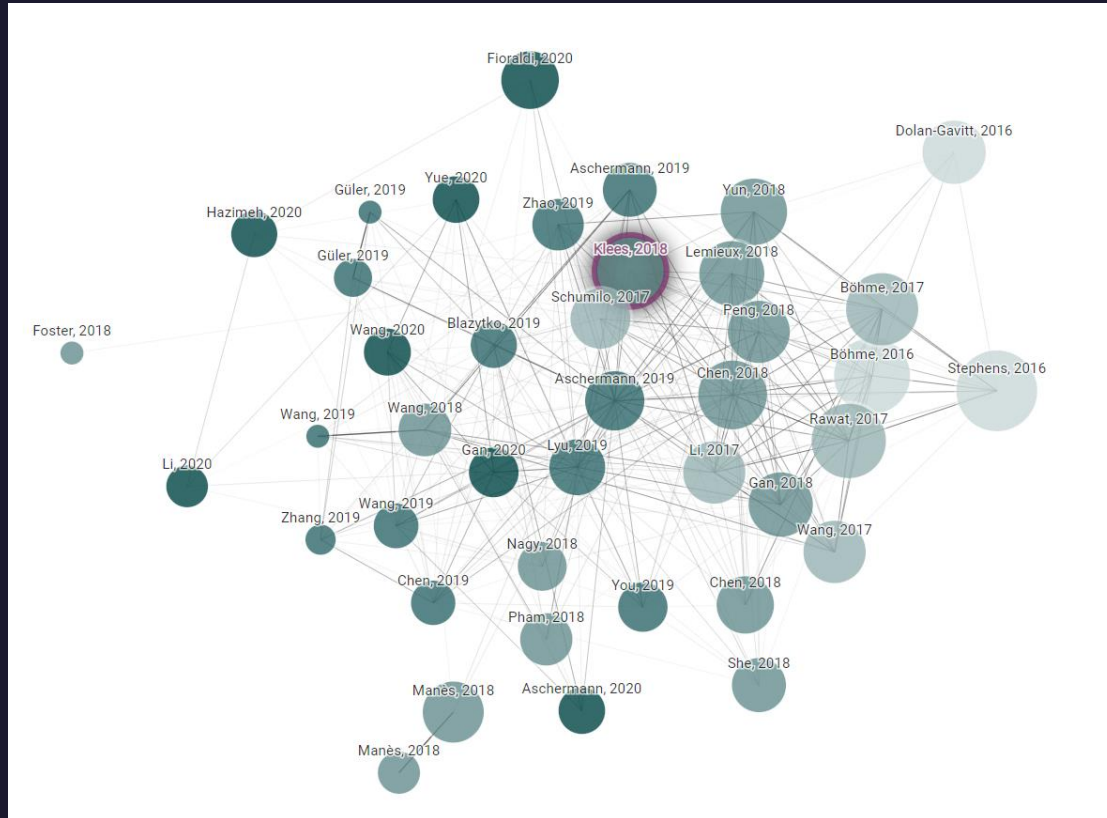
- Evaluation conclusions

- multiple trials with statistical tests to distinguish distributions
- a range of benchmark target programs with known bugs (e.g. LAVA-M, CGC, or old programs with bug fixes)
- measurement of performance in terms of known bugs, rather than heuristics based on AFL coverage profiles or stack hashes
- block or edge coverage can be used as a secondary measure
- a consideration of various (well documented) seed choices including empty seed
- timeouts of at least 24 hours, or else justification for less, with performance plotted over time.



# “Evaluating Fuzz Testing”

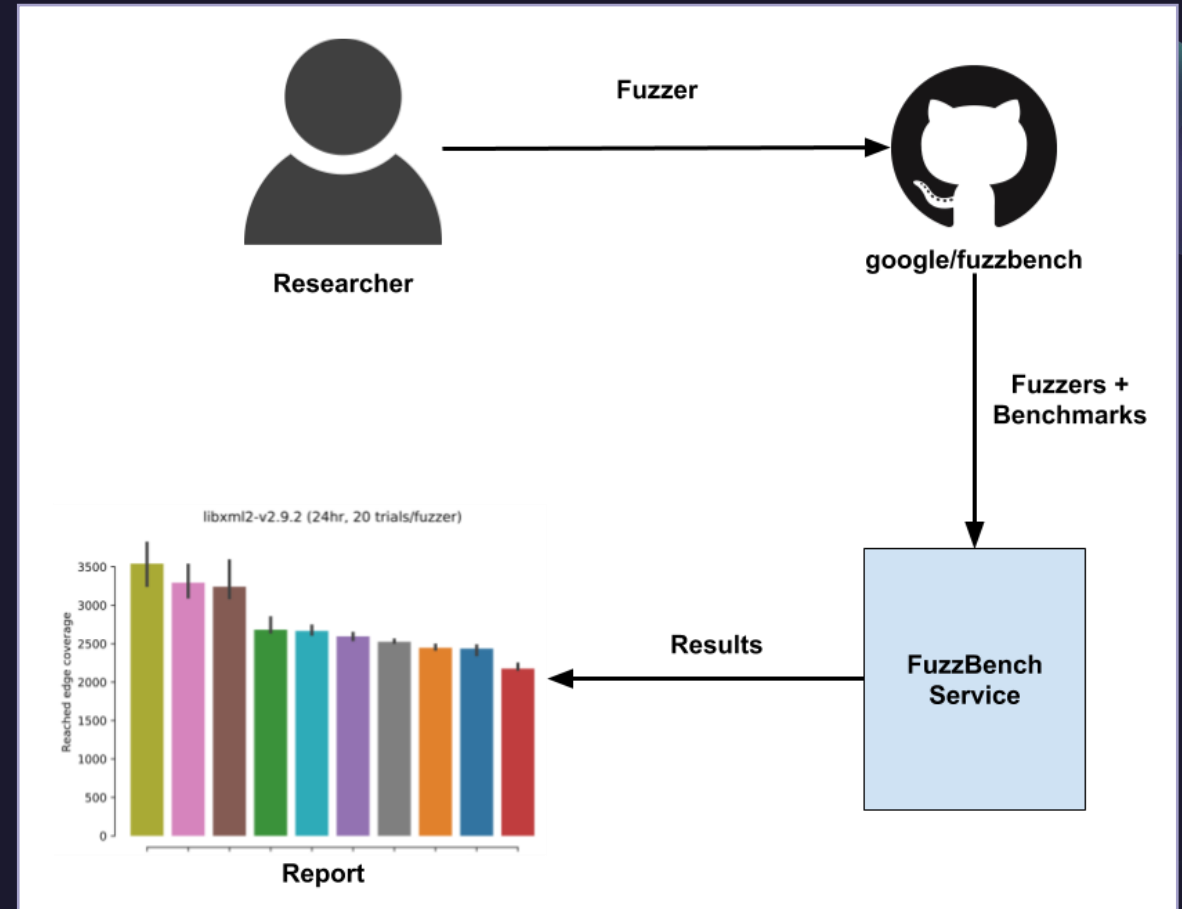
GEORGE KLEES, ANDREW RUEF, BENJI COOPER, SHIYI WEI, MICHAEL HICKS



# Benchmarking: Observable Success

## FUZZBENCH

- FuzzBench is a free resource for researchers to run batches of fuzzing runs using their work in a repeatable and comparable framework with other SoTA fuzzers



# Benchmarking: Observable Success

FUZZBENCH

## experiment summary

We show two different aggregate (cross-benchmark) rankings of fuzzers. The first is based on the average of per-benchmarks scores, where the score represents the percentage of the highest reached median code-coverage on a given benchmark (higher value is better). The second ranking shows the average rank of fuzzers, after we rank them on each benchmark according to their median reached code-coverages (lower value is better).

### By avg. score

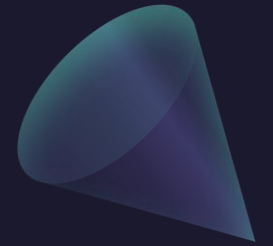
fuzzer	average normalized score
<b>aflplusplus</b>	98.39
<b>libafl</b>	97.74
<b>aflplusplusplus</b>	97.68
<b>honggfuzz</b>	96.39
<b>entropic</b>	93.79
<b>mopt</b>	86.06
<b>aflsmart</b>	85.30
<b>eclipser</b>	85.17
<b>afl</b>	85.06
<b>aflfast</b>	83.63
<b>libfuzzer</b>	81.99
<b>fairfuzz</b>	81.13
<b>centipede</b>	70.48

### By avg. rank

fuzzer	average rank
<b>aflplusplusplus</b>	2.96
<b>aflplusplus</b>	3.00
<b>libafl</b>	5.00
<b>honggfuzz</b>	5.26
<b>entropic</b>	5.74
<b>eclipser</b>	6.52
<b>aflsmart</b>	6.70
<b>afl</b>	7.52
<b>mopt</b>	7.70
<b>libfuzzer</b>	8.91
<b>centipede</b>	9.78
<b>aflfast</b>	9.87
<b>fairfuzz</b>	9.96

### experiment summary

bloaty\_fuzz\_target  
 curl\_curl\_fuzzer\_http  
 freetype2\_ftfuzzer  
 harfbuzz\_hb-shape-fuzzer  
 jsoncpp\_jsoncpp\_fuzzer  
 lcms\_cms\_transform\_fuzzer  
 libjpeg-turbo\_libjpeg\_turbo\_fuzzer  
 libpcap\_fuzz\_both  
 libpng\_libpng\_read\_fuzzer  
 libxml2\_xml  
 libxslt\_xpath  
 mbedtls\_fuzz\_dtlsclient  
 openh264\_decoder\_fuzzer  
 openssl\_x509  
 openthread\_ot-ip6-send-fuzzer  
 proj4\_proj\_crs\_to\_crs\_fuzzer  
 re2\_fuzzer  
 sqlite3\_ossfuzz  
 stb\_stbi\_read\_fuzzer  
 systemd\_fuzz-link-parser  
 vorbis\_decode\_fuzzer  
 woff2\_convert\_woff2ttf\_fuzzer  
 zlib\_zlib\_uncompress\_fuzzer  
 experiment data

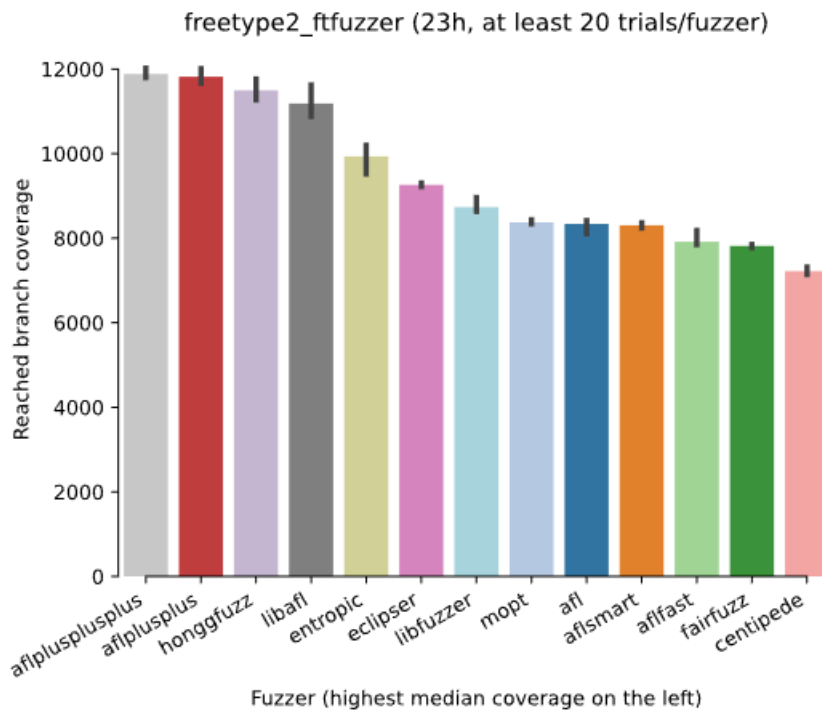


# Benchmarking: Observable Success

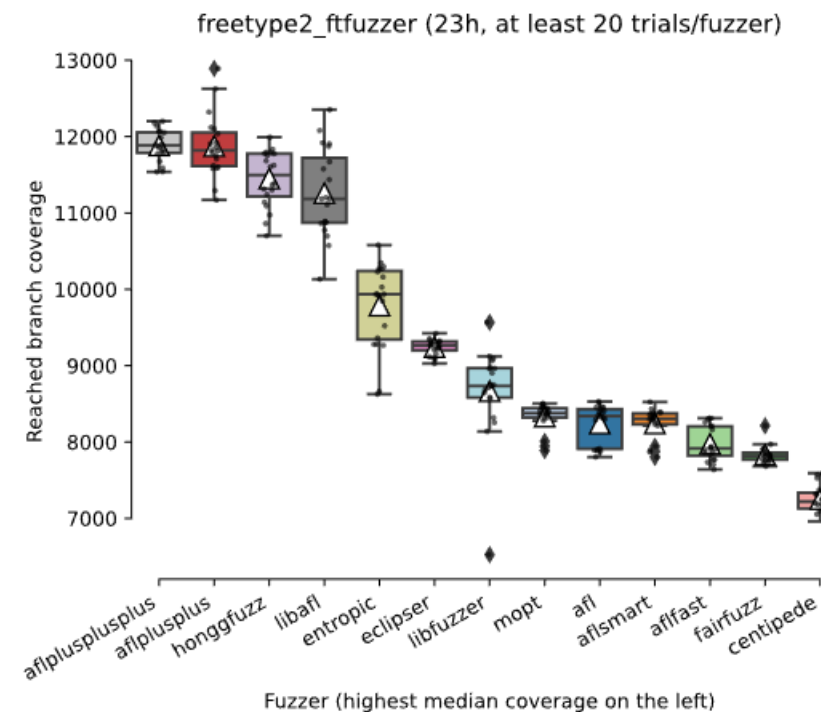
FUZZBENCH

## freetype2\_ftfuzzer summary

Ranking by median reached code coverage



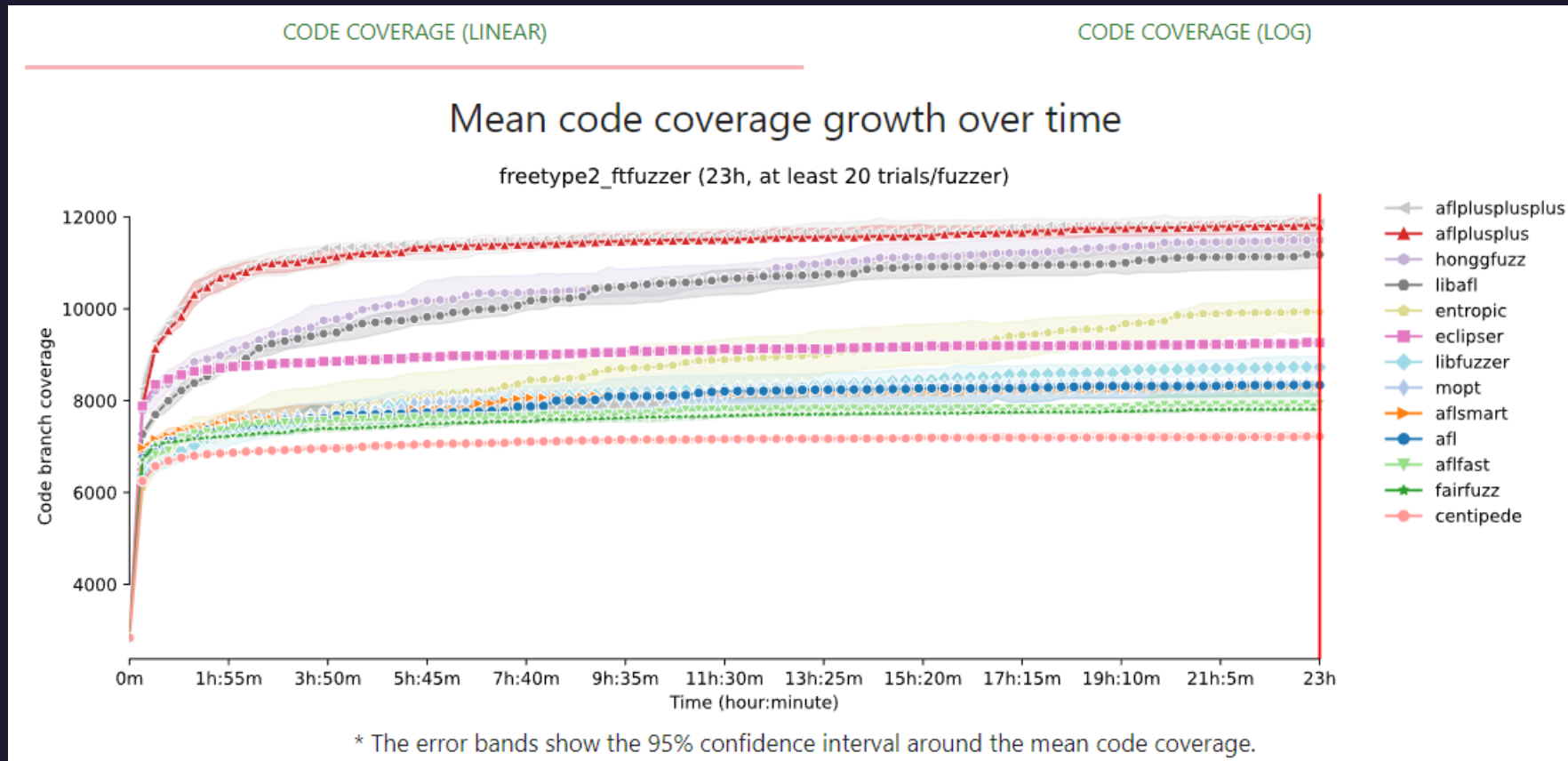
Reached code coverage distribution





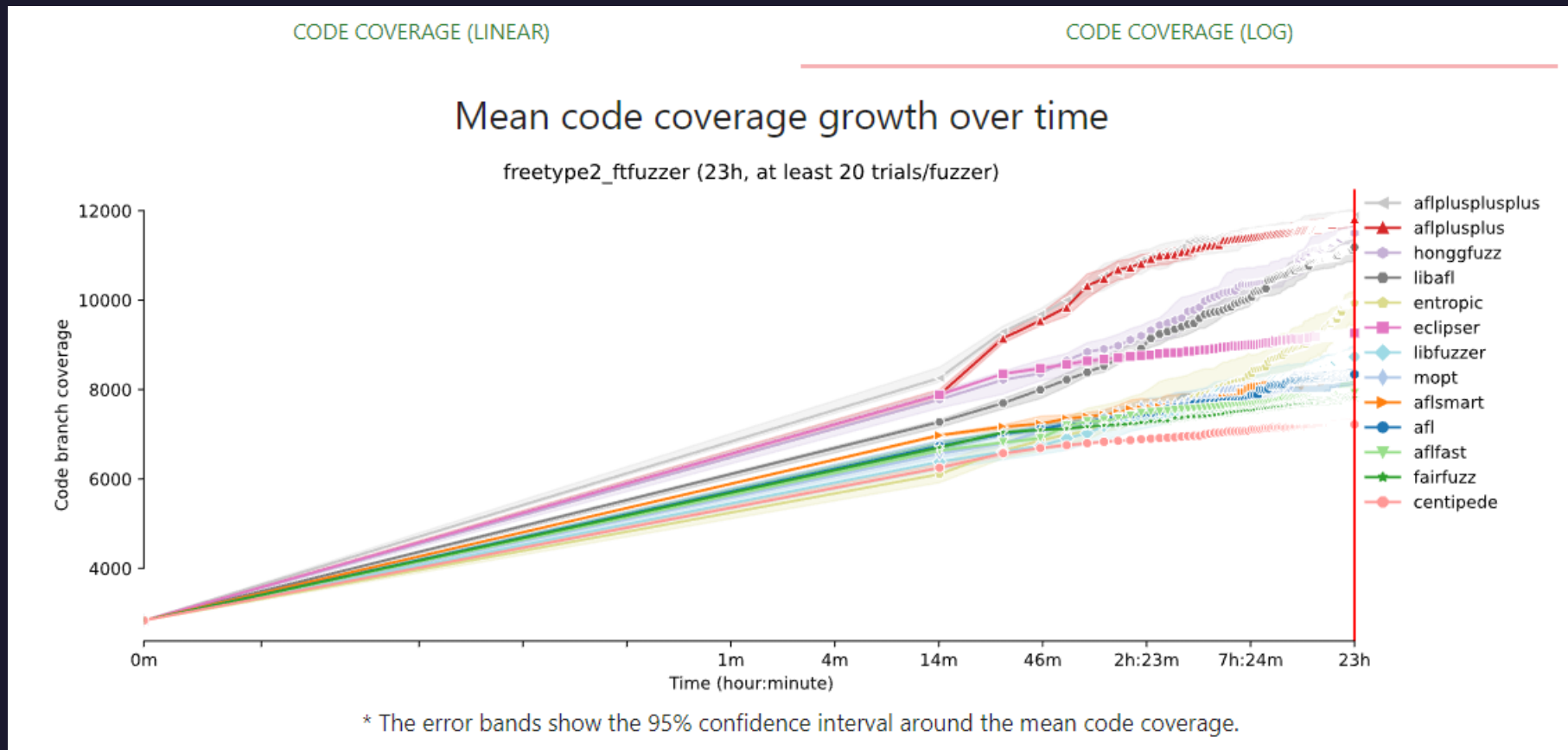
# Benchmarking: Observable Success

FUZZBENCH – COVERAGE OVER TIME VIEW



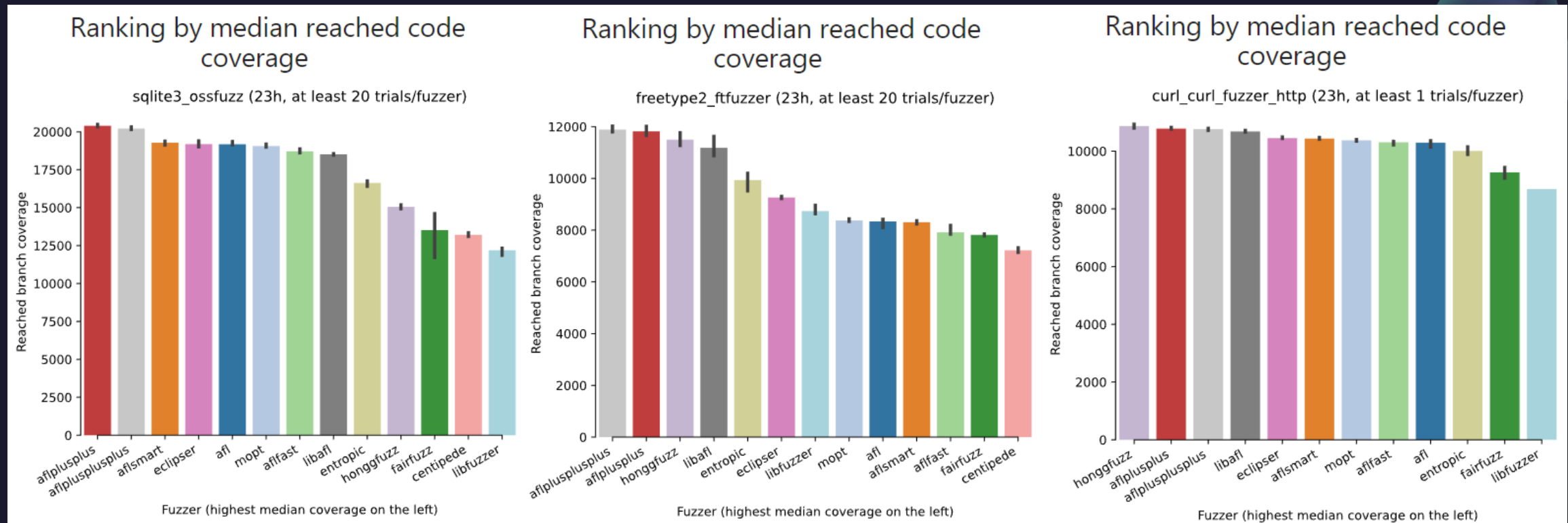
# Benchmarking: Observable Success

FUZZBENCH – LOG TIME VIEW DIFFERENTIATOR FOR LONGER FUZZING CAMPAIGNS



# Benchmarking: Observable Success

FUZZBENCH – COMPARING PERFORMANCE OF FUZZER VS PARSER



# Fuzz Introspector

LIKE GCOV REPORTS BUT FOR FUZZING!

- Fuzz introspector is a tool to help fuzzer developers to get an understanding of their fuzzer's performance and identify any potential blockers
- Integrated with OSS-FUZZ

## Project overview

### Project information

#### Reachability overview

This is the overview of reachability by the existing fuzzers in the project

	Reached	Unreached
Complexity	61.54% (10850 / 17628)	38.45% (6778 / 17628)
Functions	50.12% (1003 / 2001)	49.87% (998 / 2001)

If you implement fuzzers that target the [remaining optimal functions](#) then the reachability will be:

	Reached	Unreached
Complexity	71.27% (12565 / 17628)	28.72% (5063 / 17628)
Functions	59.47% (1190 / 2001)	40.52% (811 / 2001)

#### Fuzzers overview

Fuzzer filename	Functions Reached	Functions unreached	Fuzzer depth	Files reached	Basic blocks reached	Cyclomatic complexity	Details
/src/htslib/test/fuzz/hts_open_fuzzer.c	1003	1026	17.0	67	26998	10850	hts_open_fuzzer.c

# Fuzz Introspector

LIKE GCOV REPORTS BUT FOR FUZZING!

Project functions overview

Show  entries Search:

Func name	Git URL	Functions filename	Arg count	Args	Function reach depth	Fuzzers hit count	I Count	BB Count	Cyclomatic complexity	Functions reached	Reached by functions	Accumulated cyclomatic complexity	Undiscovered complexity
<code>LLVMFuzzerTestOneInput</code>	<a href="#">LINK</a>	/src/htslib/test/fuzz/hts_open_fuzzer.c	2	[char *, size_t]	17	0	65	16	3	1003	0	10853	3
<code>view_san</code>	<a href="#">LINK</a>	/src/htslib/test/fuzz/hts_open_fuzzer.c	1	[struct.htsFile *]	17	1	74	22	8	895	1	9493	0
<code>view_vcf</code>	<a href="#">LINK</a>	/src/htslib/test/fuzz/hts_open_fuzzer.c	1	[struct.htsFile *]	17	1	84	23	8	861	1	9005	2
<code>bam_index_build</code>	<a href="#">LINK</a>	/src/htslib/sam.c	2	[char *, int]	19	0	12	3	2	736	0	7066	196
<code>sam_index_build</code>	<a href="#">LINK</a>	/src/htslib/sam.c	2	[char *, int]	18	0	12	3	2	735	0	7064	194
<code>sam_index_build2</code>	<a href="#">LINK</a>	/src/htslib/sam.c	3	[char *, char *, int]	18	0	12	3	2	735	1	7064	194
<code>sam_index_build3</code>	<a href="#">LINK</a>	/src/htslib/sam.c	4	[char *, char *, int, int]	17	0	97	22	6	734	3	7062	192
<code>sam_write1</code>	<a href="#">LINK</a>	/src/htslib/sam.c	3	[struct.htsFile *, struct.sam_hdr_t *, struct.bam1_t *]	18	1	486	73	27	654	2	6629	10
<code>hts_close_or_abort</code>	<a href="#">LINK</a>	/src/htslib/test/fuzz/hts_open_fuzzer.c	1	[struct.htsFile *]	18	1	23	5	2	636	3	6396	10
<code>hts_close</code>	<a href="#">LINK</a>	/src/htslib/hts.c	1	[struct.htsFile *]	17	1	170	33	7	635	4	6394	10

Showing 1 to 10 of 2,001 entries Previous  2 3 4 5 ... 201 Next

# Fuzz Introspector

LIKE GCOV REPORTS BUT FOR FUZZING!

- **Fuzz Introspector** is a tool to help fuzzer developers to get an understanding of their fuzzer's performance and identify any potential blockers
- Integrated with OSS-FUZZ

## Call tree

### Function coverage

The following is the call tree with color coding for which functions are hit/not hit. This info is based on the cov

```
0 LLVMFuzzerTestOneInput [function]. [call site]
  1 malloc [call site]
  1 abort [call site]
  1 hopen [function]. [call site]
    2 find_scheme_handler [function]. [call site]
      3 isalnum_c [function]. [call site]
      4 __ctype_b_loc [call site]
      3 tolower_c [function]. [call site]
      4 __ctype_tolower_loc [call site]
      3 pthread_mutex_lock [call site]
      3 load_hfile_plugins [function]. [call site]
      4 kh_init_scheme_string [function]. [call site]
        5 calloc [call site]
      4 hfile_add_scheme_handler [function]. [call site]
        5 hts_log [function]. [call site]
```

# Fuzzer Challenges

THE CRUCIBLE FOR FUZZING AND SOLVER TOOLS

- For those who are looking to ensure their new fuzzer is able to solve difficult challenges encountered in the field, AFL++ team has created the **Fuzzer Challenges** project

## Testcases:

- `test-u8` - several chained 8 bits checks
- `test-u16` - several chained 16 bits checks
- `test-u32` - several chained 32 bits checks
- `test-u64` - several chained 64 bits checks
- `test-u128` - several chained 128 bits checks
- `test-u32-cmp` - several chained 32 bit lesser/greater checks
- `text-extint` - `llvm_ExtInt()` tests
- `test-float` - several chained float checks
- `test-double` - several chained double checks
- `test-longdouble` - several chained long double checks
- `test-memcmp` - several chained memcmp checks
- `test-strcmp` - several chained strncasecmp checks
- `test-transform` - different transforming string checks
- `test-crc32` - several chained crc32 checks

# Fuzzer Challenges

THE CRUCIBLE FOR FUZZING AND SOLVER TOOLS

- For those who are looking to ensure their new fuzzer is able to solve difficult challenges encountered in the field, AFL++ team has created the **Fuzzer Challenges** project

**TARGET** can be (currently) one of:

- AFL++
- AFL++-qemu
- AFL++-frida
- libAFL (WIP)
- honggfuzz
- libfuzzer
- symsan (via `test-symsan.sh`, use it's docker container)
- symcc + qemu (via `test-symcc.sh` and `test-symqemu.sh`)
- manticore (via `test-manticore.sh`)
- tritondse (via `test-tritondse.sh`)
- fuzzolic (via `test-fuzzolic.sh` + [docker.io/ercoppa/fuzzolic](https://docker.io/ercoppa/fuzzolic))

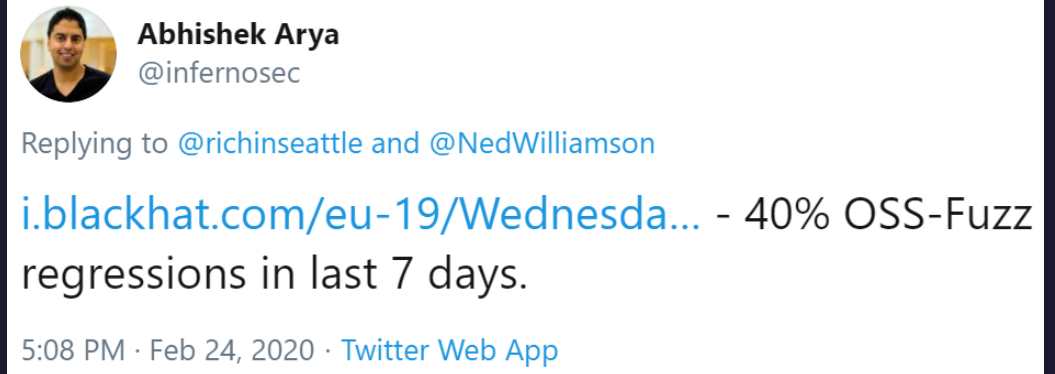


# Bugs Exist in New Code

About half of the exploitable vulnerabilities found in Chrome in 2019 were found in code less than a year old

40% of OSS-Fuzz bug finds were new check-ins before code shipped to stable

Why? Specifications rapidly change for the infrastructure that runs our businesses and code churn is high



The best defense is a good offense, which in the case for code owners is integrating continuous fuzzing in the build chain

# Building the Fuzz Chain

## FUZZING IN THE COMPILER

- LibFuzzer is now part of Clang/LLVM and Visual Studio
- Go 1.18 has integrated libFuzzer
- Rust has cargo-fuzz for integrated LibFuzzer harnesses. Supports structured fuzzing using the “Arbitrary” crate

```
// Copyright 2016 Google Inc. All Rights Reserved.
// Licensed under the Apache License, Version 2.0 (the "License");
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <assert.h>
#include <stdint.h>
#include <stddef.h>

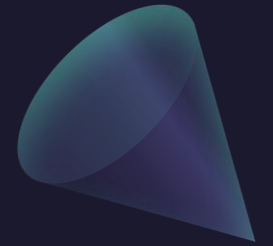
SSL_CTX *Init() {
    SSL_library_init();
    SSL_load_error_strings();
    ERR_load_BIO_strings();
    OpenSSL_add_all_algorithms();
    SSL_CTX *sctx;
    assert (sctx = SSL_CTX_new(TLSv1_method()));
    /* These two file were created with this command:
       openssl req -x509 -newkey rsa:512 -keyout server.key \
       -out server.pem -days 9999 -nodes -subj /CN=a/
    */
    assert(SSL_CTX_use_certificate_file(sctx, "runtime/server.pem",
                                       SSL_FILETYPE_PEM));
    assert(SSL_CTX_use_PrivateKey_file(sctx, "runtime/server.key",
                                       SSL_FILETYPE_PEM));
    return sctx;
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    static SSL_CTX *sctx = Init();
    SSL *server = SSL_new(sctx);
    BIO *sinbio = BIO_new(BIO_s_mem());
    BIO *soutbio = BIO_new(BIO_s_mem());
    SSL_set_bio(server, sinbio, soutbio);
    SSL_set_accept_state(server);
    BIO_write(sinbio, Data, Size);
    SSL_do_handshake(server);
    SSL_free(server);
    return 0;
}
```

# Building the Fuzz Chain

## FUZZING IN THE RUNTIME

- Jazzer – coverage guided fuzzing for JVM from **Code Intelligence**
- Wraps native LibFuzzer to fuzz Java, Kotlin, Scala, Clojure
- User creates a library with native implementation of **LLVMFuzzerTestOneInput** callback and calls Java methods via JNI
- Comes with special sanitizers for detecting injection bugs, user controlled class deserialization, etc



# Building the Fuzz Chain

## JAZZER FUZZING HARNESS

- Support for Jazzer fuzz targets is now in OSS-FUZZ and the JUnit testing framework

```
import com.code_intelligence.jazzer.api.FuzzedDataProvider;

import com.google.json.JsonSanitizer;

public class DenylistFuzzer {
    public static void fuzzerTestOneInput(FuzzedDataProvider data) {
        String input = data.consumeRemainingAsString();
        String output;
        try {
            output = JsonSanitizer.sanitize(input, 10);
        } catch (ArrayIndexOutOfBoundsException e) {
            // ArrayIndexOutOfBoundsException is expected if nesting depth is
            // exceeded.
            return;
        }
        // See https://github.com/OWASP/json-sanitizer#output.
        if (output.contains("</script") || output.contains("<script")
            || output.contains("<!--") || output.contains("]]>")) {
            System.err.println("input : " + input);
            System.err.println("output: " + output);
            throw new IllegalStateException("Output contains forbidden substring");
        }
    }
}
```

# Building the Fuzz Chain

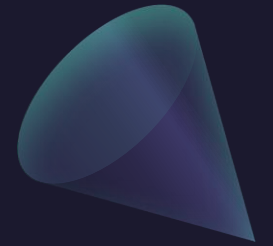
JAZZER SUPPORTS CUSTOM SANITIZERS

```
class PathTraversalSanitizer {
    @MethodHook(type = HookType.AFTER, targetClassName = "java.io.File",
targetMethod = "<init>", targetMethodDescriptor = "(Ljava/lang/String;)")
public static void fileConstructorHook(MethodHandle method, Object thisObject, Object[] arguments, int hookId, Object returnValue) {
    File file = (File) thisObject;
    String pathname = (String) arguments[0];
    try {
        // Check whether the canonical path of `file` lies inside a known list of allowed paths.
        if (!file.getCanonicalPath().startsWith("/expected/path")) {
            // If not, throw a distinctive exception that is reported by Jazzer.
            throw new PotentialPathTraversalException();
        }
    } catch(IOException e) {
    }
}
```

# Fuzzing in the Cloud

SECURITY IS A SHARED RESPONSIBILITY AND GOOGLE IS HERE TO HELP!

- OSS-FUZZ is an amazing resource. Hundreds of opensource projects receive free fuzzing with libFuzzer and AFL++ engines (or Jazzer, go fuzz, cargo fuzz, etc)
- Google is the top contributor to opensource fuzzing tooling and free compute to perform fuzz testing in the cloud
- OSS-Fuzz also offers a bounty program for writing fuzzing harnesses including rewards for security bugs found by contributed harnesses





# AFL++ - still the best general fuzzer

AFL++ - MARC HEUSE, HEIKO EISSFELDT, ANDREA FIORALDI, DOMINIK MAIER

- Persistent fuzzing, libFuzzer harness compat, inmem testcases
- LAF-Intel / CmpCov
- CmpLog / Redqueen
- Higher performance binary fuzzing
- Nyx, QEMU, Unicorn, Frida, QDBI

Feature/Instrumentation	afl-gcc	llvm_mode	gcc_plugin	qemu_mode	unicorn_mode
NeverZero	x	x(1)	(2)	x	x
Persistent mode		x	x	x86[_64]/arm[64]	x
LAF-Intel / CompCov		x		x86[_64]/arm[64]	x86[_64]/arm
CmpLog		x		x86[_64]/arm[64]	
Whitelist		x	x	(x)(3)	
Non-colliding coverage		x(4)		(x)(5)	
InsTrim		x			
Ngram prev_loc coverage		x(6)			
Context coverage		x			
Snapshot LKM support		x		(x)(5)	



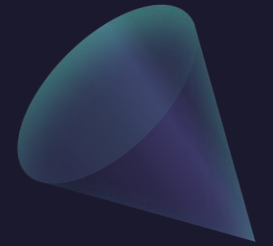


# LibAFL: Modular Fuzzer Design

LIBAFL – ANDREA FIORALDI, DOMINIK MAIER, ET AL

LibAFL gives you many of the benefits of an off-the-shelf fuzzer, while being completely customizable. Some highlight features currently include:

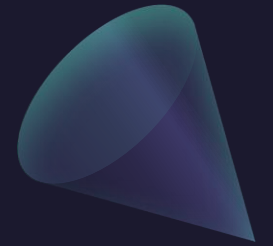
- **fast** : We do everything we can at compile time, keeping runtime overhead minimal. Users reach 120k execs/sec in frida-mode on a phone (using all cores).
- **scalable** : **Low Level Message Passing**, **LLMP** for short, allows LibAFL to scale almost linearly over cores, and via TCP to multiple machines.
- **adaptable** : You can replace each part of LibAFL. For example, **BytesInput** is just one potential form input: feel free to add an AST-based input for structured fuzzing, and more.
- **multi platform** : LibAFL was confirmed to work on *Windows, MacOS, Linux, and Android* on *x86\_64* and *aarch64*. **LibAFL** can be built in **no\_std** mode to inject LibAFL into obscure targets like embedded devices and hypervisors.
- **bring your own target** : We support binary-only modes, like Frida-Mode, as well as multiple compilation passes for sourced-based instrumentation. Of course it's easy to add custom instrumentation backends.



# LibAFL: Modular Fuzzer Design

LIBAFL – ANDREA FIORALDI, DOMINIK MAIER, ET AL

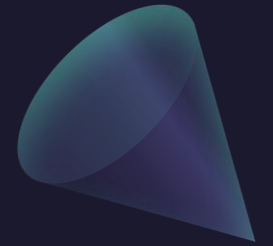
<b>corpus</b>	Corpuses contain the testcases, either in memory, on disk, or somewhere else.
<b>events</b>	Eventmanager manages all events that go to other instances of the fuzzer.
<b>executors</b>	Executors take input, and run it in the target.
<b>feedbacks</b>	The feedbacks reduce observer state after each run to a single <code>is_interesting</code> -value. If a testcase is interesting, it may be added to a Corpus.
<b>fuzzer</b>	The <code>Fuzzer</code> is the main struct for a fuzz campaign.
<b>generators</b>	Generators may generate bytes or, in general, data, for inputs.
<b>inputs</b>	Inputs are the actual contents sent to a target for each exeuction.
<b>monitors</b>	Keep stats, and display them to the user. Usually used in a broker, or main node, of some sort.
<b>mutators</b>	Mutators mutate input during fuzzing.
<b>observers</b>	Observers give insights about runs of a target, such as coverage, timing, stack depth, and more.
<b>prelude</b>	The purpose of this module is to alleviate imports of many components by adding a glob import.
<b>schedulers</b>	Schedule the access to the Corpus.
<b>stages</b>	A <code>Stage</code> is a technique used during fuzzing, working on one <code>crate::corpus::Corpus</code> entry, and potentially altering it or creating new entries. A well-known <code>Stage</code> , for example, is the mutational stage, running multiple <code>crate::mutators::Mutator</code> s against a <code>crate::corpus::Testcase</code> , potentially storing new ones, according to <code>crate::feedbacks::Feedback</code> . Other stages may enrich <code>crate::corpus::Testcase</code> s with metadata.
<b>state</b>	The fuzzer, and state are the core pieces of every good fuzzer



# LibAFL: Modular Fuzzer Design

LIBAFL – ANDREA FIORALDI, DOMINIK MAIER, ET AL

- LibAFL is a modular fuzzing system that provides mix and match components for instrumentation, data generation, and runtime configuration
- Many backends are supported - LLVM SanitizerCoverage, Frida, QEMU user and system, TinyInst, and more
- In the near future, AFL++ will be a frontend for LibAFL
- See paper from CCS'22 “LibAFL: A Framework to Build Modular and Reusable Fuzzers”





# ***Advanced Instrumentation***



# Sanitizers for Sanity

HEARTBLEED IMPACTED 66% OF WWW SERVERS

- Heartbleed taught us that fuzzers and checkers work in harmony to find deep bugs
- Today's best practices for fuzzing will include at least one worker node with sanitizers enabled.
- AddressSanitizer is only one of many



# Sanitizers for Sanity

ADDRESS SANITIZER LIFTED THE SHADOW

- Use after free (dangling ptr deref)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

```

==5740==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x62900009748 at pc 0x56371e75ade7 bp 0x7fff3c5d0410 sp 0x7fff3c5cfbe0
READ of size 55043 at 0x62900009748 thread T0
#0 0x56371e75ade6 in __asan_memcpy (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x2adde6) (BuildId: f795380dc529b46b3df17c01f9d)
#1 0x56371e7a4d45 in tls1_process_heartbeat /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/t1_lib.c:2586:3
#2 0x56371e80f8f2 in ssl3_read_bytes /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_pkt.c:1092:4
#3 0x56371e813d23 in ssl3_get_message /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_both.c:457:7
#4 0x56371e7e01ef in ssl3_get_client_hello /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_srvr.c:941:4
#5 0x56371e7dc01b in ssl3_accept /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_srvr.c:357:9
#6 0x56371e79901d in LLVMFuzzerTestOneInput /home/vulndev/fuzzer-test-suite/build/./openssl-1.0.1f/target.cc:34:3
#7 0x56371e6bf443 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x16a84) (BuildId: f795380dc529b46b3df17c01f9d)
#8 0x56371e6beb99 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x213389) (BuildId: f795380dc529b46b3df17c01f9d)
#9 0x56371e6c0389 in fuzzer::Fuzzer::MutateAndTestOne() (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x213389) (BuildId: f795380dc529b46b3df17c01f9d)
#10 0x56371e6c0f05 in fuzzer::Fuzzer::Loop(std::vector<fuzzer::SizedFile, std::allocator<fuzzer::SizedFile> >&) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x213389) (BuildId: f795380dc529b46b3df17c01f9d)
#11 0x56371e6af042 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x22bd32) (BuildId: f795380dc529b46b3df17c01f9d)
#12 0x56371e6d8d32 in main (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x22bd32) (BuildId: f795380dc529b46b3df17c01f9d)
#13 0x7fb766213d8f in __libc_start_call_main csu/./sysdeps/nptl/libc_start_call_main.h:58:16
#14 0x7fb766213e3f in __libc_start_main csu/./csu/libc-start.c:392:3
#15 0x56371e6a3a84 in _start (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x16a84) (BuildId: f795380dc529b46b3df17c01f9d)

0x62900009748 is located 0 bytes to the right of 17736-byte region [0x62900005200,0x62900009748)
allocated by thread T0 here:
#0 0x56371e75babe in malloc (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x2aeabe) (BuildId: f795380dc529b46b3df17c01f9d)
#1 0x56371e843a1b in CRYPTO_malloc /home/vulndev/fuzzer-test-suite/build/BUILD/crypto/mem.c:308:8
#2 0x56371e81543c in freelist_extract /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_both.c:708:12
#3 0x56371e81543c in ssl3_setup_read_buffer /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_both.c:770:10
#4 0x56371e815a6c in ssl3_setup_buffers /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_both.c:827:7
#5 0x56371e7dccbe in ssl3_accept /home/vulndev/fuzzer-test-suite/build/BUILD/ssl/s3_srvr.c:292:9
#6 0x56371e79901d in LLVMFuzzerTestOneInput /home/vulndev/fuzzer-test-suite/build/./openssl-1.0.1f/target.cc:34:3
#7 0x56371e6bf443 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x16a84) (BuildId: f795380dc529b46b3df17c01f9d)
#8 0x56371e6c06a0 in fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::vector<fuzzer::SizedFile, std::allocator<fuzzer::SizedFile> >&) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x213389) (BuildId: f795380dc529b46b3df17c01f9d)
#9 0x56371e6c0cf2 in fuzzer::Fuzzer::Loop(std::vector<fuzzer::SizedFile, std::allocator<fuzzer::SizedFile> >&) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x213389) (BuildId: f795380dc529b46b3df17c01f9d)
#10 0x56371e6af042 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x22bd32) (BuildId: f795380dc529b46b3df17c01f9d)
#11 0x56371e6d8d32 in main (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x22bd32) (BuildId: f795380dc529b46b3df17c01f9d)
#12 0x7fb766213d8f in __libc_start_call_main csu/./sysdeps/nptl/libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/vulndev/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x2adde6) (BuildId: f795380dc529b46b3df17c01f9d)
Shadow bytes around the buggy address:
 0x0c527fff9290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c527fff92a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c527fff92b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c527fff92c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c527fff92d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c527fff92e0: 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa
0x0c527fff92f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c527fff9300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c527fff9310: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c527fff9320: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c527fff9330: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  
```

# Sanitizers for Sanity

## LLVM SANITIZER FAMILY

- The LLVM Sanitizer family has grown to include:
  - **Undefined Behavior Sanitizer**
  - Hardware-assisted Address Sanitizer
  - Thread Sanitizer (races and deadlocks)
  - Memory Sanitizer (uninitialized memory)

## UNDEFINED BEHAVIOR SANITIZER

- Array subscript out of bounds, where the bounds can be statically determined
- Bitwise shifts that are out of bounds for their data type
- Dereferencing misaligned or null pointers
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination





# Sanitizers for Sanity

## LLVM SANITIZER FAMILY

- The LLVM Sanitizer family has grown to include:
  - Undefined Behavior Sanitizer
  - **Hardware-assisted Address Sanitizer**
  - Thread Sanitizer (races and deadlocks)
  - Memory Sanitizer (uninitialized memory)

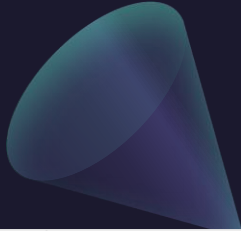
## HARDWARE-ASSISTED ADDRESS SANITIZER

- ASAN uses bitmask for shadow memory and redzone for overflows, and quarantines for UAF
- HWASAN uses hardware memory tagging
- AArch64 uses top-byte-ignore (TBI) pointer tagging
- x86\_64 will use Linear Address Masking (LAM) or page aliasing. Linear addresses use 48bit (L4) or 57bit (L5) addresses, leaving room for tagging.
- Some detections become probabilistic


# Sanitizers for Sanity

## LLVM SANITIZER FAMILY

- Sanitizers are now supported in:
- Clang 3.1+, GCC 4.8+
- Visual Studio 2019+
- LibFuzzer, AFL++, honggfuzz
- Kernels: KASAN
- Emulators: QEMU, Frida



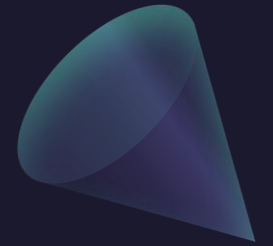
OS	x86	x86_64	ARM	ARM64	MIPS	MIPS64	PowerPC	PowerPC64
Linux	yes	yes			yes	yes	yes	yes
OS X	yes	yes						
iOS Simulator	yes	yes						
FreeBSD	yes	yes						
Android	yes	yes	yes	yes				



# Unconstrained Progress

WHEN YOU CAN'T BEAT THEM, UNJOIN THEM

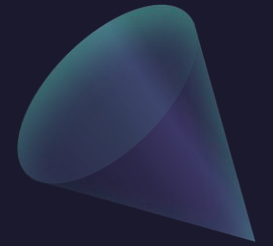
- Code logic is structured as a series of constraints or comparisons that decide which code path to follow
- For a long time, it was assumed constraint solvers were the solution
- Constraint solvers require **heavy analysis** including **lifting** target **code** to intermediate languages to **extract the semantics** of each instruction. Solving requires converting the series of instructions to Boolean algebra in a format compatible with SMTLIB2 or domain specific encoding – **computationally expensive** for quick iteration



# Unconstrained Progress

WHEN YOU CAN'T BEAT THEM, UNJOIN THEM

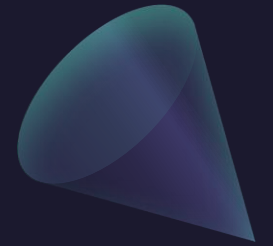
- LAF-Intel was the first fuzzing technology to approach the problem of constraints through code transformation
- Multibyte comparisons are rewritten as a series of sequential single-byte comparisons.
- 32bit:  $1/(2^{32}) \rightarrow 1/(2^8 * 4)$ . **4,294,967,296 vs 1024**
- CmpCov is the updated version of LAF-Intel seen in fuzzers today



# Focused Mutation

CHEAP ALTERNATIVE TO TAINT TRACKING AND SYMBOLIC EXECUTION

- RedQueen was the first fuzzing technology to efficiently correlate values observed in comparisons as being derived directly from the input
- **Solves magic bytes and checksums** by spending time fuzzing input byte offsets seen in the comparison instructions **without needing taint tracking or symbolic execution**
- CmpLog is the updated implementation seen in fuzzing tools today



# Improved Input Generation

A network diagram consisting of white circular nodes connected by thin white lines, set against a dark blue background. The nodes are arranged in a complex, interconnected pattern, resembling a molecular structure or a data network. The lines are thin and white, while the nodes are slightly larger and also white. The overall aesthetic is clean and technical.

**“Grammar mutators are able to trigger deep bugs that are near impossible to find with code coverage guided fuzzers”**  
vanHauser, Berlin 2022

```
ctx.rule: adds a rule NONTERM->RHS. We can use {NONTERM} in the RHS to request a recursion.
ctx.rule("START", "<document>{XML_CONTENT}</document>")
ctx.rule("XML_CONTENT", "{XML}{XML_CONTENT}")
#ctx.script: adds a rule NONTERM->func(*RHS).
# In contrast to normal `rule`, RHS is an array of nonterminals. It's up to the function to
# combine the values returned for the NONTERMINALS with any fixed content used.
ctx.script("XML", ["TAG", "ATTR", "XML_CONTENT"], lambda tag,attr,body: b"<%s %s>%s</%s>"%
(tag,attr,body,tag) )
ctx.rule("ATTR", "foo=bar")
ctx.rule("TAG", "some_tag")
#ctx.regex: limit applying a rule for mutations
# sometimes we don't want to explore the set of possible inputs in more detail. For example,
# if we fuzz a script interpreter, we don't want to spend time on fuzzing all different
# variable names. In such cases we can use Regex terminals. Regex terminals are only mutated
# during generation, but not during normal mutation stages, saving a lot of time.
# The fuzzer still explores different values for the regex, but it won't be able to learn
# interesting values incrementally.
# Use this when incremental exploration would most likely waste time.
ctx.regex("TAG", "[a-z]+")
```

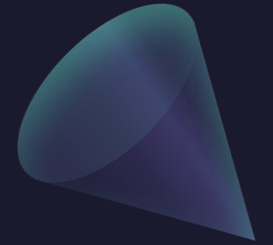
# Grammars

## CONTEXT FREE GRAMMARS

- DEFENSICS
- Domato
- Nautilus
- GrammaTron
- AFL++ ATNWalk
- Fzero / F1 Fuzzer

## STRUCTURED / API GRAMMARS

- Peach (AFLSmart)
- libprotobuf-mutator
- Fuzzili
- SyzLang
- KernelFuzz

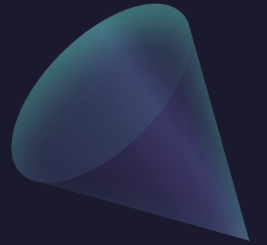




# Searching for Approximate Grammar

FUZZING IS A CHEAP GRAMMAR EXTRACTION

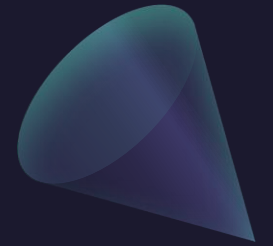
- It has long been known that high quality **grammars** can accurately **model syntax and interaction logic** to perform testing
- Grammars are tedious to produce, and **implementations may deviate** from the specification
- The simplest improvement to bit flipping is to use a dictionary of tokens
- **DICT2FILE** is a LLVM plugin that ships with AFL++ that can extract constant comparisons at compile time. Generating a dictionary can be as simple as that!



# Grammars are a Browser's Best Friend

DOM FUZZING WITH DOMATO - IVAN FRATRIC / GOOGLE

- A few simple components can compromise a browser...
- `grammar.py` - Generic BNF parser and generative fuzzer
- `*.txt`, `*.html` - HTML/CSS/js grammar and `.html` template
- `generator.py` - Test case generator for DOM fuzzing



# Grammars are a Browser's Best Friend

DOM FUZZING WITH DOMATO - IVAN FRATRIC / GOOGLE

Domato uses an eBNF inspired context-free grammar

```
$ python generator.py --output_dir out --no_of_files 3
```

```
Running on ClusterFuzz
```

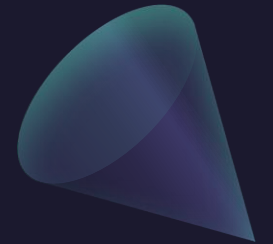
```
Output directory: out
```

```
Number of samples: 3
```

```
Writing a sample to out\fuzz-0.html 08/30/2022 07:40 PM 455,255 fuzz-0.html
```

```
Writing a sample to out\fuzz-1.html 08/30/2022 07:40 PM 463,901 fuzz-1.html
```

```
Writing a sample to out\fuzz-2.html 08/30/2022 07:40 PM 460,684 fuzz-2.html
```



# Grammars are a Browser's Best Friend

DOM FUZZING WITH DOMATO – IVAN FRATRIC / GOOGLE

Domato uses an eBNF inspired context-free grammar

```
<cssproperty_z-index> = <fuzzint>
```

```
<cssproperty_z-index> = auto
```

```
<cssproperty_z-index> = inherit
```

```
<cssproperty_zoom> = <percentage>%
```

```
<cssproperty_zoom> = <fuzzint>
```

```
<cssproperty_zoom> = <float>
```

```
<cssproperty_zoom> = auto
```

```
<cssproperty_zoom> = reset
```



# Grammars are a Browser's Best Friend

DOM FUZZING WITH DOMATO – IVAN FRATRIC / GOOGLE

To inject lines of code into the **grammar**, a special syntax is used

```
!begin lines
```

```
<new element> = document.getElementById("<string min=97 max=122>");
```

```
<element>.doSomething();
```

```
!end lines
```

This will expand types as well as generate new variable names



# Grammars are a Browser's Best Friend

DOM FUZZING WITH DOMATO – IVAN FRATRIC / GOOGLE

To inject lines of code into the **generator**, another syntax is used

```
!begin function createbody
  n = int(context['size'])
  ret_val = 'a' * n
!end function
```

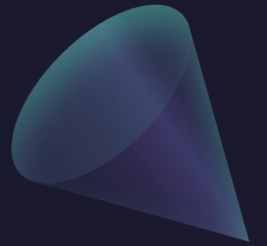
```
<body> = <call function=createbody>
```



# Grammars are a Browser's Best Friend

JAVASCRIPT JIT FUZZING - FUZZILI - SAMUEL GROB

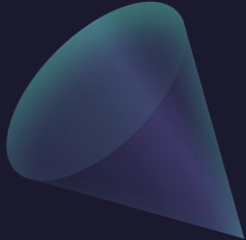
- To fuzz the Chrome V8 JavaScript JIT, Samuel created a custom intermediate language that represents dependencies and mutations
- Fuzzilli has found vulnerabilities in every JavaScript engine with a JIT
- The key to Fuzzilli is that I can lift JavaScript source from disk to IL and instrument the AST prior to the JIT engine having access to the code
- JIT vulns have been the primary attack vector on browsers for 5 years
- An Intermediate Language fuzzer is a form of grammar driven mutator



# Grammars are a Browser's Best Friend

## LIBPROTOBUF-MUTATOR VS THE CHROME SANDBOX

- Ned Williamson showed that APIs can be encoded as ProtoBuf definitions and found Chrome sandbox vulns in “Attacking Chrome IPC” (2019)
- LibProtoBufMutator allows users to describe basic types and structures and use metaprogramming for fuzzing



```
message MyFormat {
  oneof a_or_b {
    MessageA message_a = 1;
    MessageB message_b = 2;
  }
  required MessageC message_c = 3;
}
```



# Grammars for APIs

## LIBPROTOBUF-MUTATOR VS APIS

- libprotobuf-mutator is generalizable to any type of API fuzzing and is integrated directly with LibFuzzer.
- OSS-FUZZ has example targets using libprotobuf-mutator to deserialize mutated buffers

```
// Needed since we use getenv().
#include <stdlib.h>

// Needed since we use std::cout.
#include <iostream>

#include "testing/libfuzzer/proto/lpm_interface.h"

// Assuming the .proto file is path/to/your/proto_file/my_format.proto.
#include "path/to/your/proto_file/my_format.pb.h"

// Put your conversion code here (if needed) and then pass the result to
// your fuzzing code (or just pass "my_format", if your target accepts
// protobufs).

DEFINE_PROTO_FUZZER(const my_fuzzer::MyFormat& my_proto_format) {
    // Convert your protobuf to whatever format your targeted code accepts
    // if it doesn't accept protobufs.
    std::string native_input = convert_to_native_input(my_proto_format);

    // You should provide a way to easily retrieve the native input for
    // a given protobuf input. This is useful for debugging and for seeing
    // the inputs that cause targeted_function to crash (which is the reason we
    // are here!). Note how this is done before targeted_function is called
    // since we can't print after the program has crashed.
    if (getenv("LPM_DUMP_NATIVE_INPUT"))
        std::cout << native_input << std::endl;

    // Now test your targeted code using the converted protobuf input.
    targeted_function(native_input);
}
```

# Grammars for Syscalls

## GRAMMAR FUZZING SYSCALLS

- System Calls are “just” a privileged API between userland and the kernel

### Syscall Description Language (syzlang)

```
open(file ptr[in, filename], flags flags[open_flags]) fd  
read(fd fd, buf ptr[out, array[int8]], count bytesize[buf])  
close(fd fd)
```

```
open_flags = O_RDONLY, O_WRONLY, O_RDWR, O_APPEND
```

# When in Doubt, Math it Out

HIGHLY SELECTIVE APPLICATION OF HIGHLY SOPHISTICATED TECH WINS IN THE END

- FuzzBench and other empirical data has deflated the hype around symbolic execution, taint analysis, and other complex analysis
- In a compute constrained environment, it's typically more efficient to use naive high throughput approaches like fuzzing
- Selective use of graph analysis to locate “frontiers” that are guarded by complex constraints can identify when to apply more complex approaches to code exploration

# When in Doubt, Math it Out

SYMCC / SYMQEMU

- SYMCC / SYMQEMU are SoTA implementations of concolic execution that can work on large code bases
- Integration exists in AFL++ via custom mutators
- Less than 1% of edges require symex

```
define i32 @is_double(i32, i32) {  
    ; symbolic computation  
    %3 = call i8* @_sym_get_parameter_expression(i8 0)  
    %4 = call i8* @_sym_get_parameter_expression(i8 1)  
    %5 = call i8* @_sym_build_integer(i64 1)  
    %6 = call i8* @_sym_build_shift_left(i8* %4, i8* %5)  
    %7 = call i8* @_sym_build_equal(i8* %6, i8* %3)  
    %8 = call i8* @_sym_build_bool_to_bits(i8* %7)  
  
    ; concrete computation (as before)  
    %9 = shl nsw i32 %1, 1  
    %10 = icmp eq i32 %9, %0  
    %11 = zext i1 %10 to i32  
  
    call void @_sym_set_return_expression(i8* %8)  
    ret i32 %11  
}
```

# When in Doubt, Math it Out

SYMCC / SYMQEMU

- SYMCC is based on an LLVM instrumentation and can be efficient for source based targets
- However if source is available then other compiler plugins can extract grammars which will be more efficient than using constraint solving
- SymQemu is derivative of the SymCC implementation and works on unmodified binaries and has been shown to be more robust than alternatives

# When in Doubt, Math it Out

## SYMSAN

- Builds on top of the built in DataFlowSanitizer in LLVM
- Low runtime state management overhead due to efficient storage of high level types
- 2x faster than SymCC/SymQEMU
- Minimal memory overhead

## TRITON-DSE

- High level API for Triton from Quarkslab uses a unique internal graph representation that is highly efficient
- Available as an extension to AFL++ but not proven to be more capable than SymQEMU

# When in Doubt, Math it Out

## TRITON-DSE

- “TritonDSE goal is to provide higher-level primitives than Triton. Triton is a low-level framework where one have to provide manually all instructions to be executed symbolically.”

Loader mechanism (based on LIEF, cle or custom ones)

Memory segmentation

Coverage strategies (block, edge, path)

Pointer coverage

Automatic input injection on stdin, argv

Input replay with QBDI

input scheduling (*customizable*)

sanitizer mechanism

basic heap allocator

some libc symbolic stubs



# ***Reaching New Attack Surface***



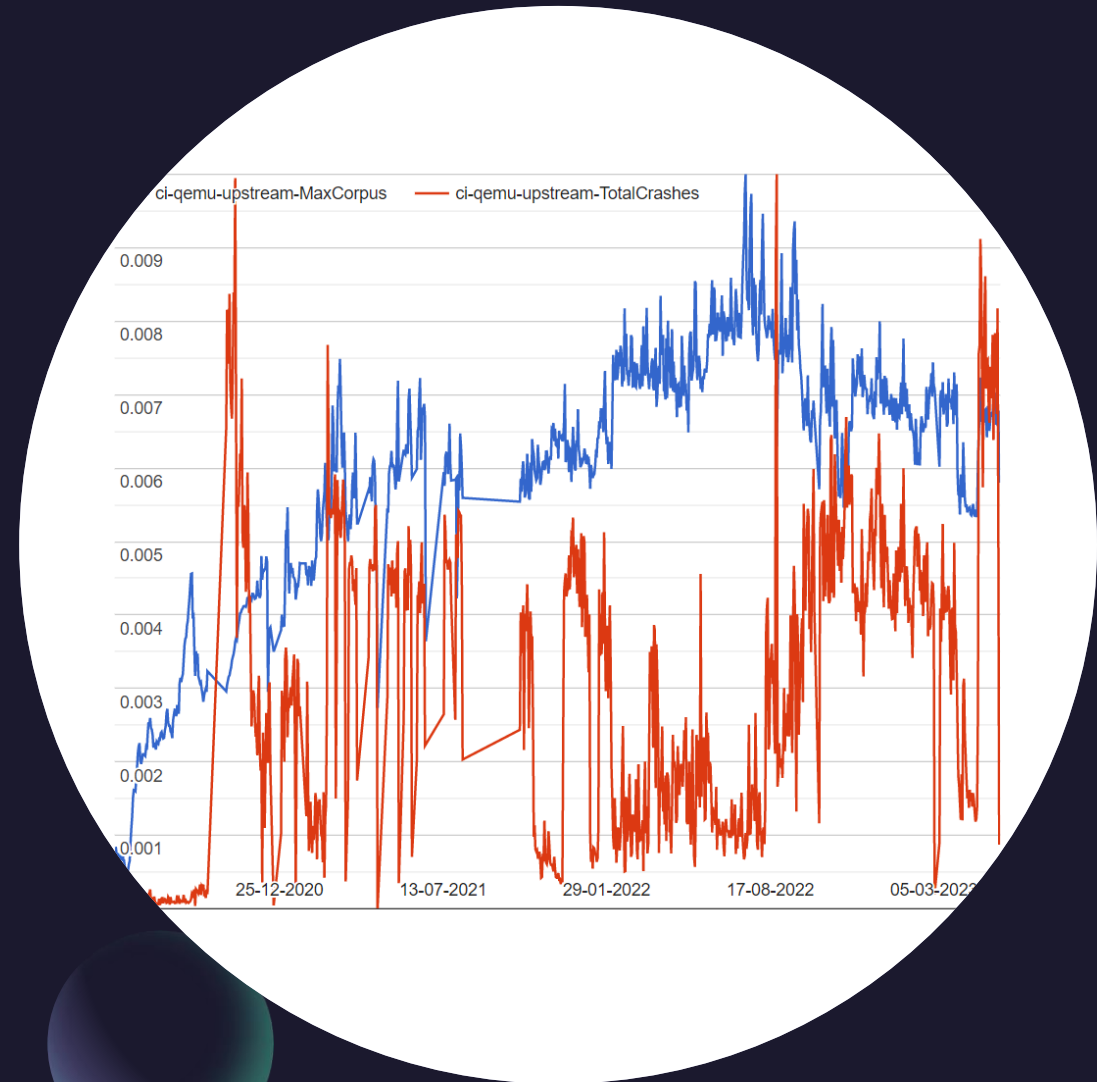
**2019: 2046 Linux  
kernel bugs found &  
fixed by SyzKaller**  
(BlueHat IL 2020)

**2023: 4535 fixed**

**Keeping average of  
2-3/day**

SyzBot Dashboard

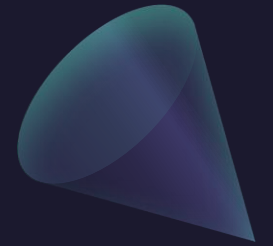
<https://syzkaller.appspot.com/upstream>



# Fuzzing Windows

## WINDOWS – THE ANTI-POSIX ENVIRONMENT

- Fuzzing on Windows has many architectural considerations
  - No forking – Address Space Randomization causes inconsistencies across runs
  - Multi-threaded by default
  - Locked access to system resources
  - An array of IPC options
  - Async behavior via COM interfaces
  - Handle based access to subsystems that need to be managed appropriately across iterations



# Fuzzing Windows with DBI

MANY CODE COVERAGE ENGINES HAVE PROPAGATED TO WINDOWS

- The solution for fuzzing windows is either to embed fuzzing in the process using a native JIT for x64->x64 or to use snapshot-oriented fuzzing based on hypervisors or emulation
- Dynamic Binary Instrumentation achieves the first option
  - DynamoRIO
  - Frida
  - Custom JIT via Debugger APIs like TinyInst

# Snapshot Fuzzing

A SNAPSHOT IN TIME UNLOCKS THE FUTURE POTENTIAL

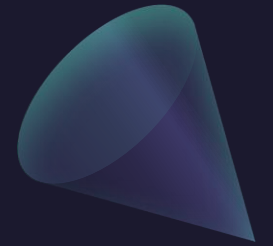
- Snapshot fuzzing solves the problems with persistent in-memory fuzzing faced by solutions like LibFuzzer, AFL++ Persistence, DBI based fuzzing, stateful exploration, and many other challenges
- For the uninitiated, snapshot fuzzing restores state between iterations
- Each modified page after a checkpoint defined by the fuzzer or initial state saved will be restored from the saved or initial state
- Solves all problems with global state management, process creation, etc



# Snapshot Fuzzing

A SNAPSHOT IN TIME UNLOCKS THE FUTURE POTENTIAL

- Snapshot fuzzing is not easy-mode and requires expert users
- Workflows are more challenging; initialization of the fuzzing loop is from a runtime state requiring configuration of virtual or hypervisor CPU state
- Snapshots can be generated incrementally to reliably handle and target state transitions
- Snapshots can be migrated to scalable environments



# Fuzzing Anything With Emulators

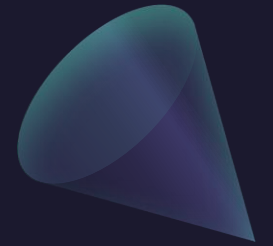
EMULATE CODE YOU CAN ISOLATE FROM HARDWARE I/O

- On the graph of performance vs targetability, emulators are the maximum for targetability
- Given enough effort anything can be simulated (ref: “the simulation”)
- In reality, there is an ideal compromise between instrumentation of an “in situ” execution environment and emulation
- Emulation can limit access to external devices, environmental config, etc

# WTF Fuzzer

SO YOU WANT TO FUZZ A WINDOWS KERNEL DRIVER

- Fuzzing closed source bare metal OS environments is challenging
- The options are to take full system memory snapshots, which will only have the kernel and a single process mapped into memory or to develop a custom hypervisor
- WTF implements the first approach – fuzzing a kernel via a kernel memory snapshot
- My team expanded this to support any OS that runs in QEMU



# WTF Fuzzer

SO YOU WANT TO FUZZ A WINDOWS KERNEL DRIVER

- In the “simple” case, a kernel debugger attaches to the runtime state at the point of the call you want to test
- Given the right amount of effort, you can hook any API and respond accordingly
- You can see this in the example multi-packet TLV sample
- Confirmed this in eBPF fuzzing we did last year – multi IOCTL

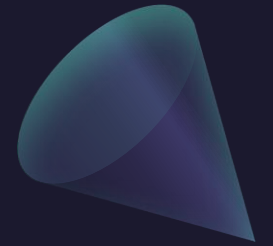




# kAFL / NYX Fuzzer

YOU WANT KASAN AND SYZKALLER ON WINDOWS?

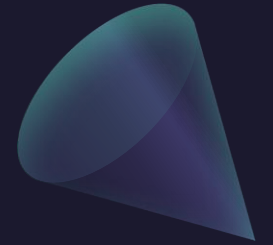
- Pure emulation based kernel fuzzing will find its limits quickly due to device I/O
- The final boss for OS fuzzing is writing a custom hypervisor that resets live memory and execution state in an OS that still has access to virtualized devices
- Using native hypervisor allows use of IntelPT for 5% full system coverage
- Achieved after a 4 year effort from RUB-SysSec in German



# kAFL / NYX Fuzzer

YOU WANT KASAN AND SYZKALLER ON WINDOWS?

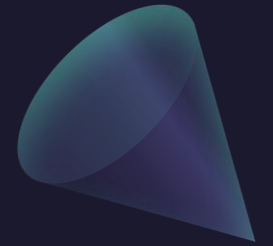
- kAFL is now forked by Intel and is state of the art for live OS fuzzing
- Nyx today is provided as an API
- Nyx is integrated in AFL++ for fuzzing hypervisor guests!



# “GDB Fuzzing”

TRADING OFF PERFORMANCE TO GET THE JOB DONE

- Anything implementing a gdb stub can technically be fuzzed
- JTAG, embedded devices, simulators, hypervisors, etc
- Recently being explored by Andreas Zeller's group in the ISSTA'23 paper “Fuzzing Embedded Systems Using Debug Interfaces” (GDBFuzz)
- Avatar<sup>2</sup> for firmware that can't be fuzzed in isolation and needs to emulate code while still accessing hardware I/O on embedded devices



# Differential Fuzzing

ASK AN ORACLE WHEN HARNESSING IS IMPOSSIBLE

- Differential Fuzzing is the answer when bugs can be found via a round trip through an encoder/decoder
- Differential Fuzzing also allows comparing across implementations of a parser. This is used by tools like SiliFuzz to fuzz CPUs or others to fuzz smart contracts

# Differential Fuzzing

ASK AN ORACLE WHEN HARNESSING IS IMPOSSIBLE

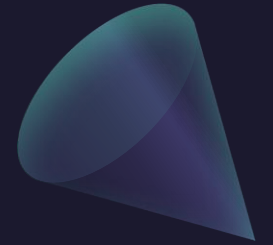
- Smart Contract Fuzzing

For inputs, there are three notable things we'll want to vary:

- The config which determines how the VM should execute (what features and such)
- The BPF program to be executed, which we'll generate like we do in "smart"
- The initial memory of the VMs

Once we've developed our inputs, we'll also need to think of our outputs:

- The "return state", the exit code itself or the error state
- The number of instructions executed (e.g., did the JIT program overrun?)
- The final memory of the VMs

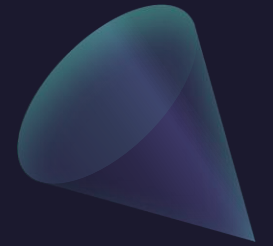


# Differential Fuzzing

ASK AN ORACLE WHEN HARNESSING IS IMPOSSIBLE

Then, to execute both JIT and the interpreter, we'll take the following steps:

- The same steps as the first fuzzers:
  - Use the rBPF verification pass (called “check”) to make sure that the VM will accept the input program
  - Initialise the memory, the syscalls, and the entrypoint
  - Create the executable data
- Then prepare to perform the differential testing
  - JIT compile the BPF code (if it fails, fail quietly)
  - Initialise the interpreted VM
  - Initialise the JIT VM
  - Execute both the interpreted and JIT VMs
  - Compare return state, instructions executed, and final memory, and panic if any do not match.





# Summary

Fuzzing can go anywhere, test anything, achieve new heights, we are only at the beginning of expanding the impact and reach of fuzzing. There is room in all three areas I focused on, but we are continually growing capabilities and reach for fuzzing

# Thank You, WOOT'23 Questions?

Richard Johnson | [rjohnson@fuzzing.io](mailto:rjohnson@fuzzing.io) | [@richinseattle](https://twitter.com/richinseattle)



**FUZZING/IO**  
Security Automation • Vulnerability Research

<https://fuzzing.io/woot23.pdf>