

# PkgFuzz Project: Yet Another Continuous Fuzzing for Open Source Software

Yuhei Kawakoya/ Yuto Otsuki/ Eitaro Shioji

2024  
**CODEBLUE**  
BECAUSE SECURITY MATTERS



# Who We Are ?

## Yuhei Kawakoya

- NTT Security Holdings
- Distinguished Researcher, Ph.D. (Engineering)
- Shifting (left ?) from malware analysis to vulnerability discovery.



## Yuto Otsuki

- NTT Security Holdings
- Senior Researcher, Ph.D. (Engineering)
- Interested in memory analysis, reverse engineering, and OS security.



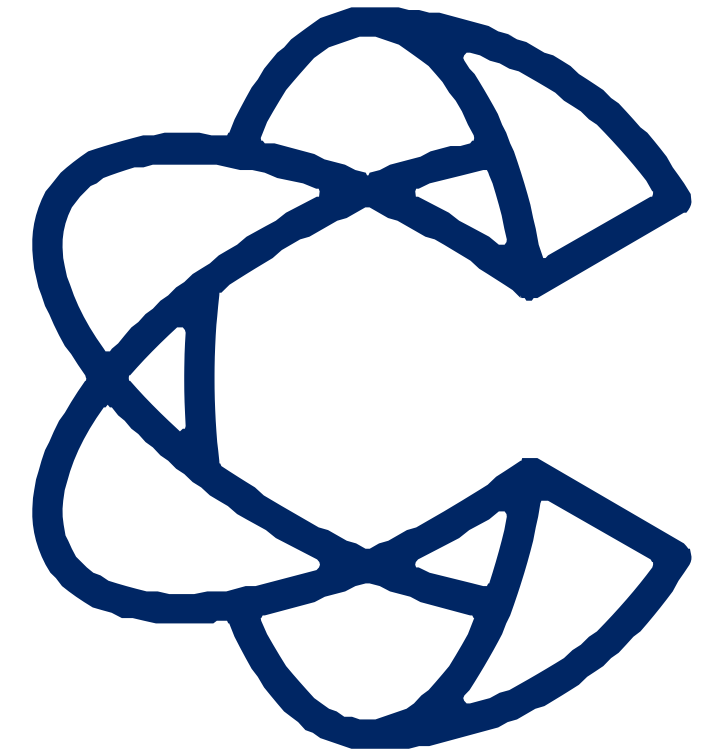
## Eitaro Shioji

- NTT Social Informatics Laboratories
- Senior Researcher
- Interested in vulnerability discovery and web/mobile/IoT security.



# Agenda

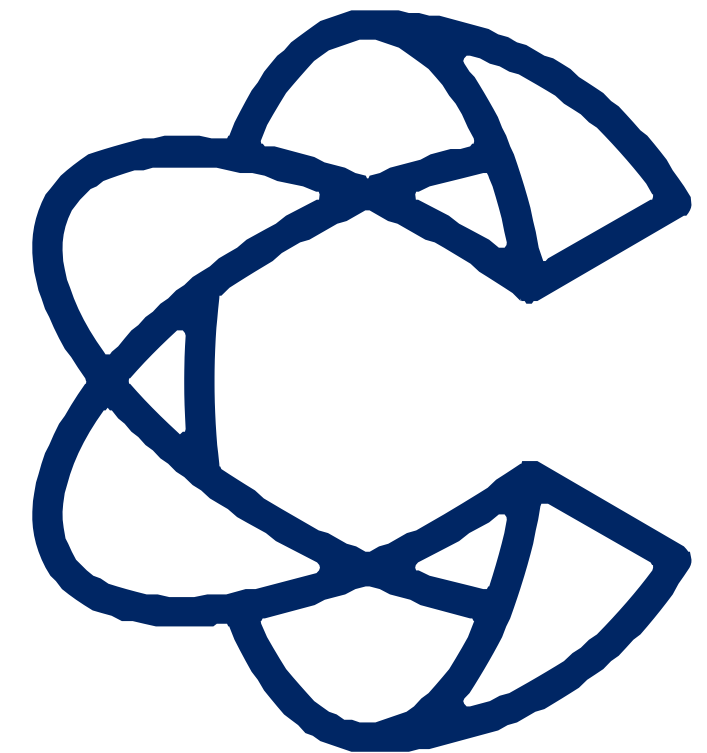
- **Vulnerability and Fuzzing**
- **PkgFuzz**
- **PkgFuzz Project**
- **From Crash to Exploit**
- **Next Step**
- **Wrap up**



# Agenda

## ● Vulnerability and Fuzzing

- PkgFuzz
- PkgFuzz Project
- From Crash to Exploit
- Next Step
- Wrap up



# Value of Vulnerabilities

- Vulnerability information, especially zero-day vulnerabilities on platforms with a large number of users, is highly valuable.
  - Bug Bounty Program
  - Exploit Acquisition Program
- The total prize money for the hacking contest Pwn2Own, which focuses on zero-day vulnerabilities, continues to increase every year.
  - 2014 – 2017: \$460,000 to \$850,000
  - 2018 – 2023: \$1,000,000 to \$2,000,000

(\*1) <https://www.zerodium.com/program.html>

(\*2) <https://www.zerodayinitiative.com/blog/2024/3/21/pwn2own-vancouver-2024-day-two-results>



In Zerodium's Exploit Acquisition Program, Windows RCE (remotely executable exploit) can be valued at up to \$1 million.\*1

A banner for the 'MASTER OF PWN' contest featuring a stylized figure of a person jumping over a hurdle.

		PRIZE \$	POINTS
1	Manfred Paul	\$202,500	25
2	Synacktiv	\$200,000	20
3	Seunghyun Lee	\$145,000	15

At Pwn2Own Vancouver 2024, the winner received a prize of \$202,500.\*2



# How to Find Vulnerabilities

- AI analysis of summaries from the past 7 years' Black Hat USA Briefings -

Approach	Explanation	Example
Fuzzing	It is an automated method for discovering vulnerabilities by sending invalid or unexpected inputs to software to cause crashes or abnormal behavior. Sources indicate that fuzzing is used on various targets such as browsers, mobile applications, network protocols, and embedded devices.	2018 The Finest Penetration Testing Framework for Software-Defined Networks
Reverse Engineering	It involves analyzing code to understand the internal workings of devices and software. Sources indicate that reverse engineering is used for firmware analysis, decoding custom protocols, and understanding hardware security mechanisms.	2018 Exploitation of a Modern Smartphone Baseband
Known Vulnerability Analysis	It involves researching known vulnerabilities, exploits, and attack techniques to discover similar vulnerabilities. Sources indicate that vulnerability analysis can be a valuable resource for identifying new vulnerabilities and attack vectors.	2018 Return of Bleichenbacher's Oracle Threat (ROBOT)
Side-channel Attacks	It exploits side-channel information, such as device performance or power consumption, to obtain sensitive data. Sources indicate that side-channel attacks can pose a serious threat, especially to embedded devices and hardware security modules.	2019 Dragonblood: Attacking the Dragonfly Handshake of WPA3
Code Audit	This method involves security analysts manually reviewing source code to identify vulnerabilities. Sources indicate that manual code audits are often used to uncover more complex vulnerabilities that are difficult to detect with automated methods like fuzzing.	2019 Lessons From Two Years of Crypto Audits

# Fuzzing 101

## ■ What is Fuzzing ?

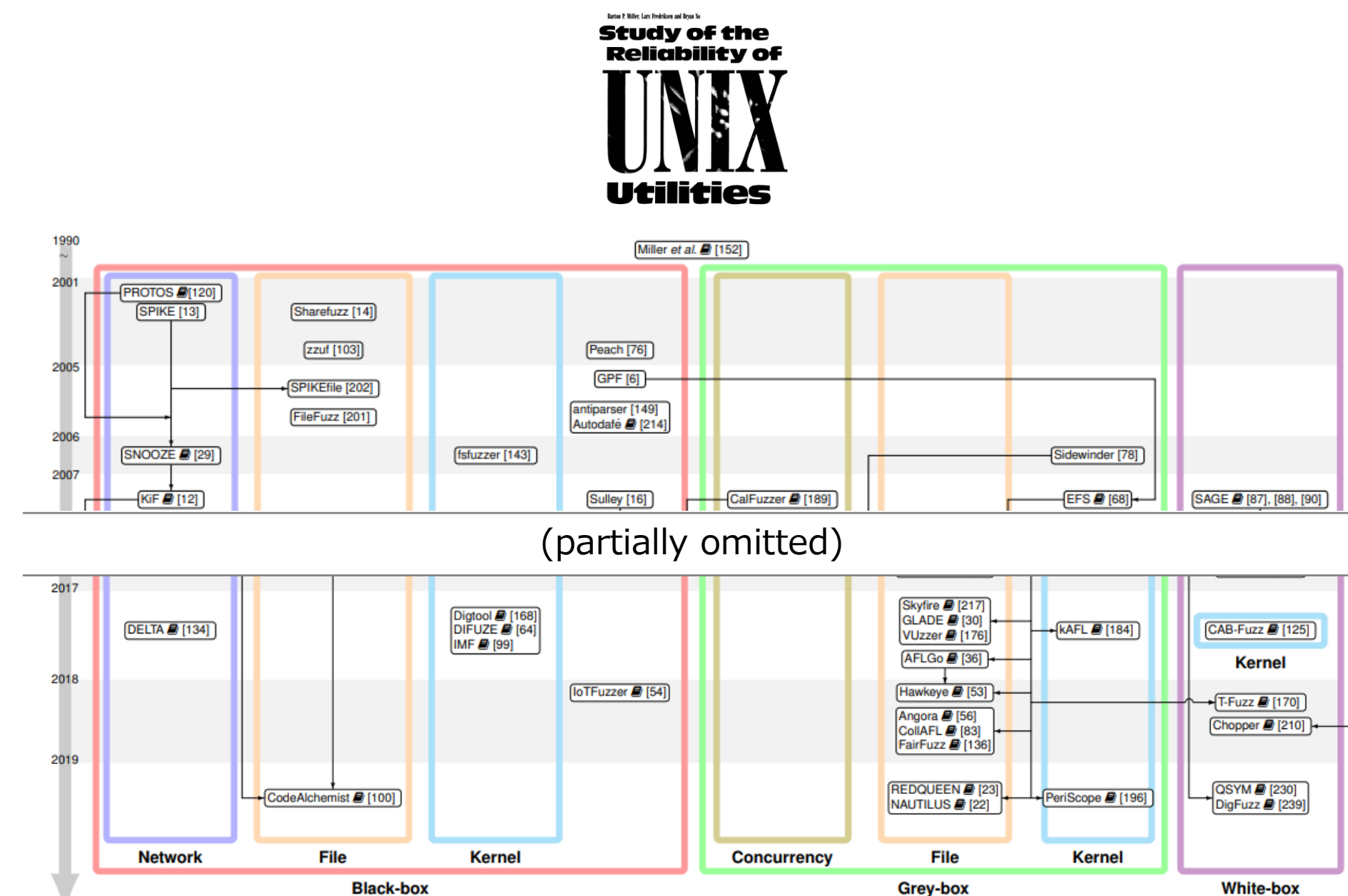
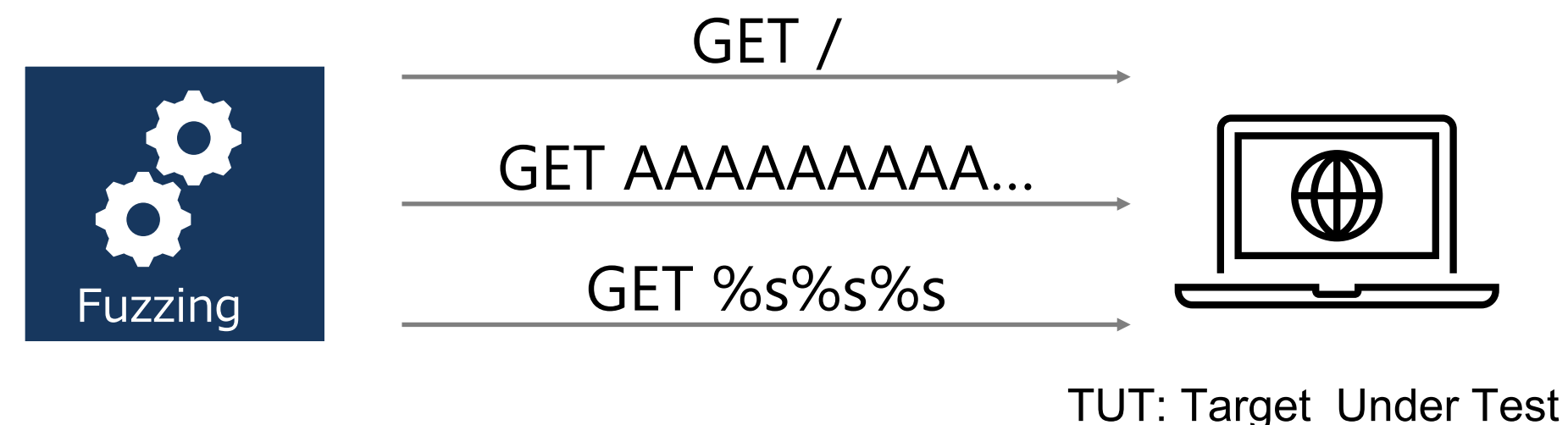
- A testing method that involves providing random values to the target program and repeatedly executing it to automatically identify inputs that cause the program to crash.

## ■ History

- *"An empirical study of the reliability of UNIX utilities"*, Communications of the ACM, 1990

## ■ Barton P. Miller

- Corrupted values, caused by communication failures due to a storm, were entered into various UNIX commands, leading to numerous crashes.
- Since his paper, many studies and fuzzers have been published.



"The Art, Science, and Engineering of Fuzzing: A Survey", Valentin J.M. Manes et al.  
<https://arxiv.org/pdf/1812.00140>

# Fuzzing Project: Google OSS-Fuzz

## ■ Overview

- A fuzzing project for OSS
- The latest fuzzing technology and a scalable cloud environment.
- Leverages community power to enable fuzzing across a wide range of software.
- Over 1,000 vulnerability fix and 36,000 bug found at August 2023<sup>\*1</sup>.

oss-fuzz oss-fuzz New issue All issues Q Type=Bug-Security label:clusterfuzz-status:Duplicate,WontFix Sign in

1 - 100 of 11736 Next List Grid Chart

ID	Type	Component	Status	Proj	Reported	Owner	Summary + Labels
935	Bug-Security	----	Verified	freetype2	2017-03-23	----	freetype2: Heap-buffer-overflow in t1_builder_add_point ClusterFuzz Reproducible
938	Bug-Security	----	Verified	ffmpeg	2017-03-25	----	ffmpeg: Global-buffer-overflow in ff_acelp_interpolatef ClusterFuzz Reproducible
939	Bug-Security	----	Verified	ffmpeg	2017-03-25	----	ffmpeg: Global-buffer-overflow in ff_h264_filter_mib_fast ClusterFuzz Reproducible
941	Bug-Security	----	Verified	freetype2	2017-03-25	----	freetype2: Heap-buffer-overflow in psh_glyph_init ClusterFuzz Reproducible
949	Bug-Security	----	Verified	libxml2	2017-03-27	----	libxml2: Heap-buffer-overflow in xmlFAParsePosCharGroup ClusterFuzz Reproducible
950	Bug-Security	----	Verified	file	2017-03-27	----	file: Heap-buffer-overflow in cdf_getuint32 ClusterFuzz Reproducible
956	Bug-Security	----	Verified	libreoffice	2017-03-28	----	libreoffice: Container-overflow in sdr::table::TableLayouter::SetBorder ClusterFuzz Reproducible
957	Bug-Security	----	Verified	grpc	2017-03-28	----	grpc: Heap-use-after-free in post_batch_completion ClusterFuzz Reproducible
958	Bug-Security	----	Verified	file	2017-03-29	----	file: Heap-buffer-overflow in cdf_read_property_info ClusterFuzz Reproducible
961	Bug-Security	----	Verified	grpc	2017-03-30	----	grpc: Heap-buffer-overflow in server_filter_incoming_metadata ClusterFuzz Reproducible
962	Bug-Security	----	Verified	libmspub	2017-03-30	----	libmspub: Container-overflow in libvenge::RVNGStringStreamPrivate::RVNGStringStreamPrivate ClusterFuzz Reproducible

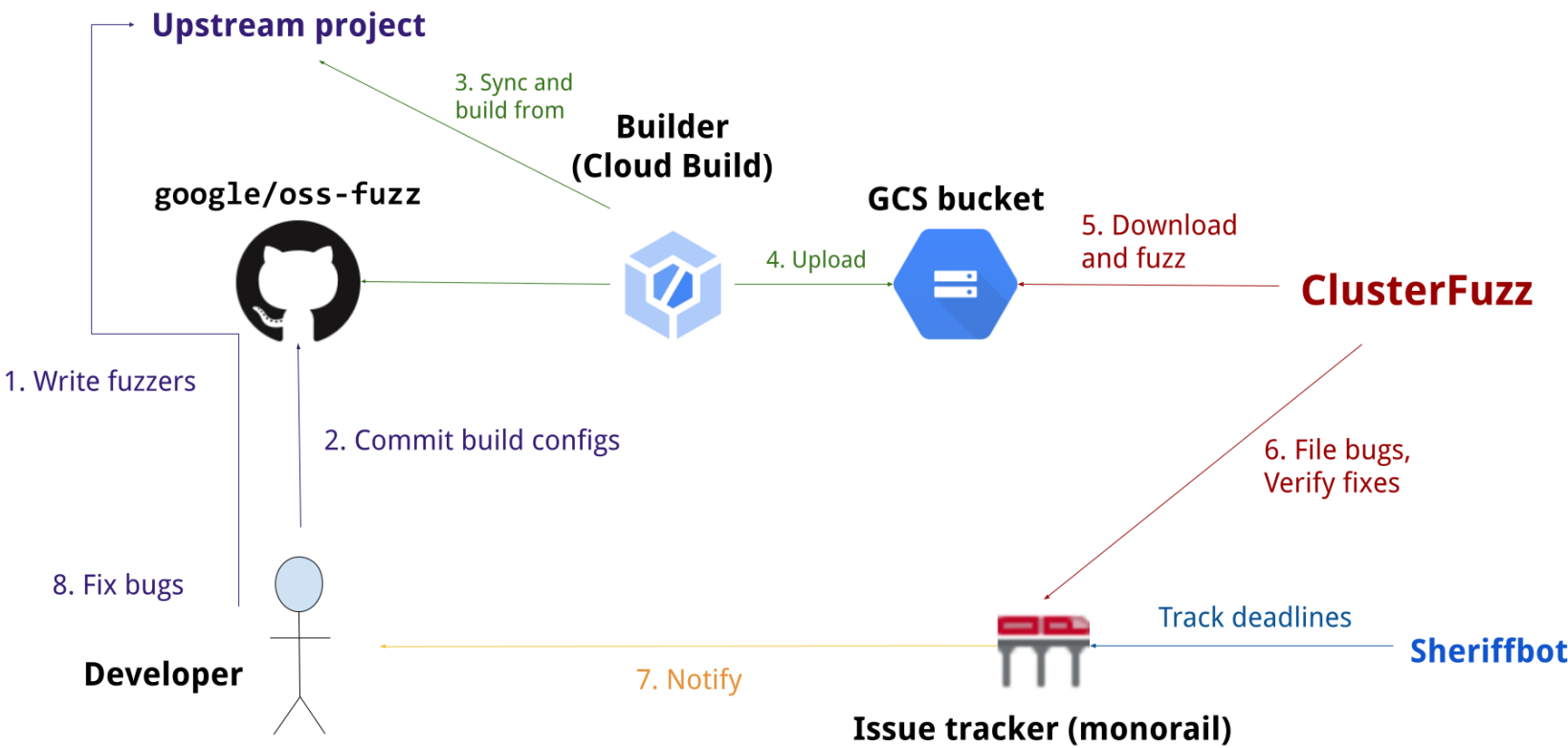
Issue Tracker (monorail)<sup>\*3</sup>

## ■ Participation Requirements

- “We accept established projects that have a critical impact on infrastructure and user security”
- Approx. 1200+ projects

## ■ Challenges

- **Fuzzing preparations totally depend on each OSS.**
- ...



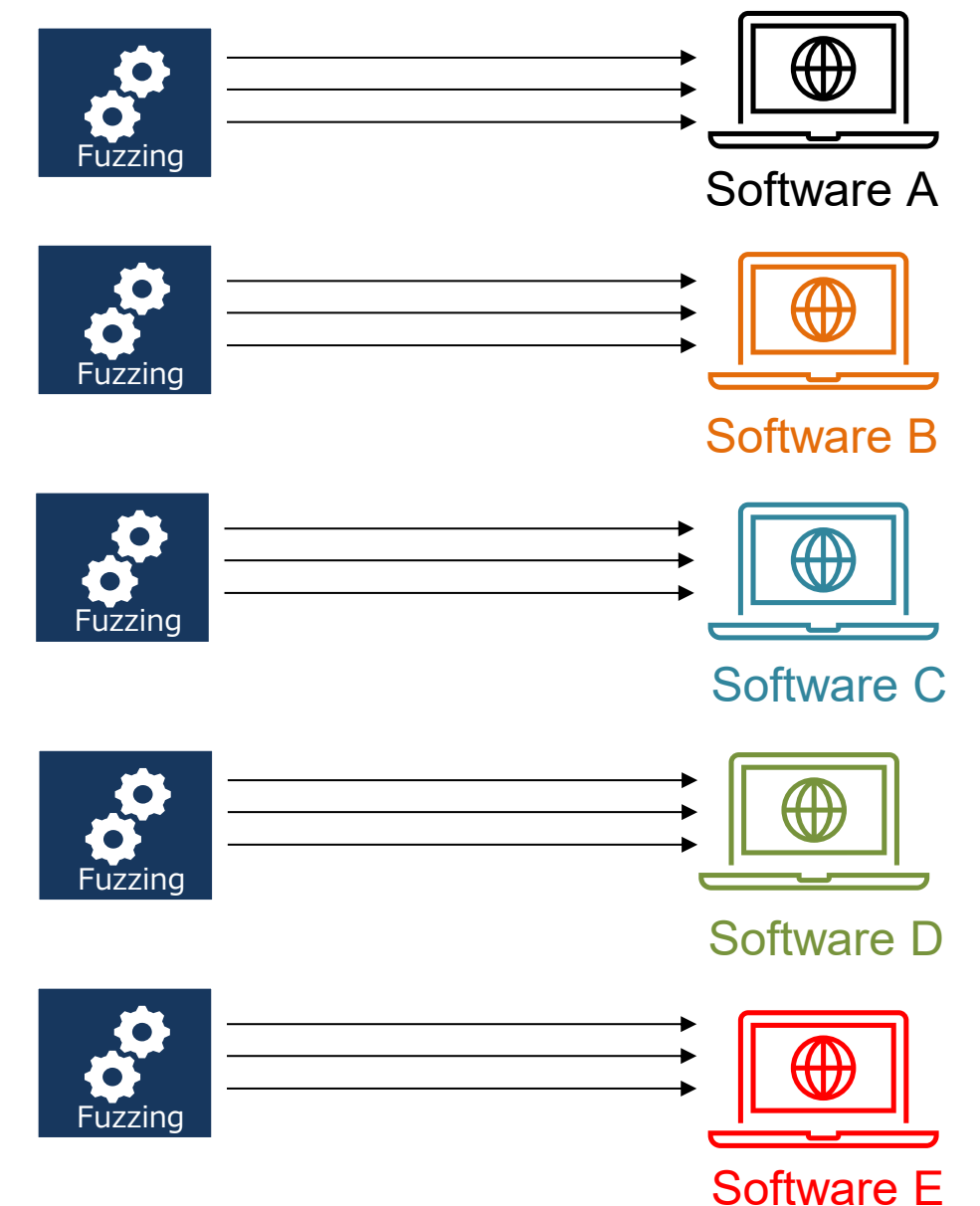
ClusterFuzz: OSS-Fuzz Infrastructure<sup>\*2</sup>

(\*1) <https://github.com/google/oss-fuzz>  
(\*2) <https://github.com/google/clusterfuzz>  
(\*3) <https://issues.oss-fuzz.com/issues?q=status:open>



# Automation Blocker for Fuzzing

- Fuzzing does not necessarily automate the entire workflow of it.
  - It's necessary for people to properly prepare in advance.
- Tasks where human intervention is needed to (efficiently) perform fuzzing.
  - **#1 Harness Preparation**
  - #2 Command-line Arguments Analysis
  - #3 Building Binaries for Fuzzing
  - **#4 Initial Seed Files**
- How OSS-Fuzz deal with them?
  - Each OSS project individually prepare them.
- If solved, what would make you happy ?
  - We can bring the benefits and impact of fuzzing to OSS projects that cannot allocate sufficient resources for security testing (fuzzing tests), primarily smaller OSS projects.



When continuously performing fuzzing on different software, human intervention decreases its efficiency.

# Automation Blocker:

## #1 Harness Preparation

- A driver for invoking the fuzzing target.
- A harness is not necessary
  - Target: main
  - \$ afl-fuzz ... ./readelf @@
- A harness is necessary
  - Target: a library function
    - E.g., `zip_file_open`
  - Since it cannot be executed directly as a program, prepare harness, e.g., `zziptest.c`, to call the library function.
    - \$ afl-fuzz ... ./zziptest @@

(\*1) <https://github.com/paroj/ZZIPlib>

(\*2) <https://github.com/bminor/binutils-gdb/blob/master/binutils/readelf.c>

binutils-gdb / binutils / readelf.c

Code

Blame

24269 lines (21350 loc) · 640 KB

```
24227
24228     int
24229     main (int argc, char ** argv)
24230     {
24231         int err;
24232
24233         #ifdef HAVE_LC_MESSAGES
24234             setlocale (LC_MESSAGES, "");
24235         #endif
24236         setlocale (LC_CTYPE, "");
24237         bindtextdomain (PACKAGE, LOCALEDIR);
24238         textdomain (PACKAGE);
24239
24240         expandargv (&argc, &argv);
24241
24242         parse_args (&cmdline, argc, argv);
```

An example that a harness is not necessary<sup>\*2</sup>

zziplib / bins / zziptest.c

Code

Blame

186 lines (165 loc) · 5.05 KB

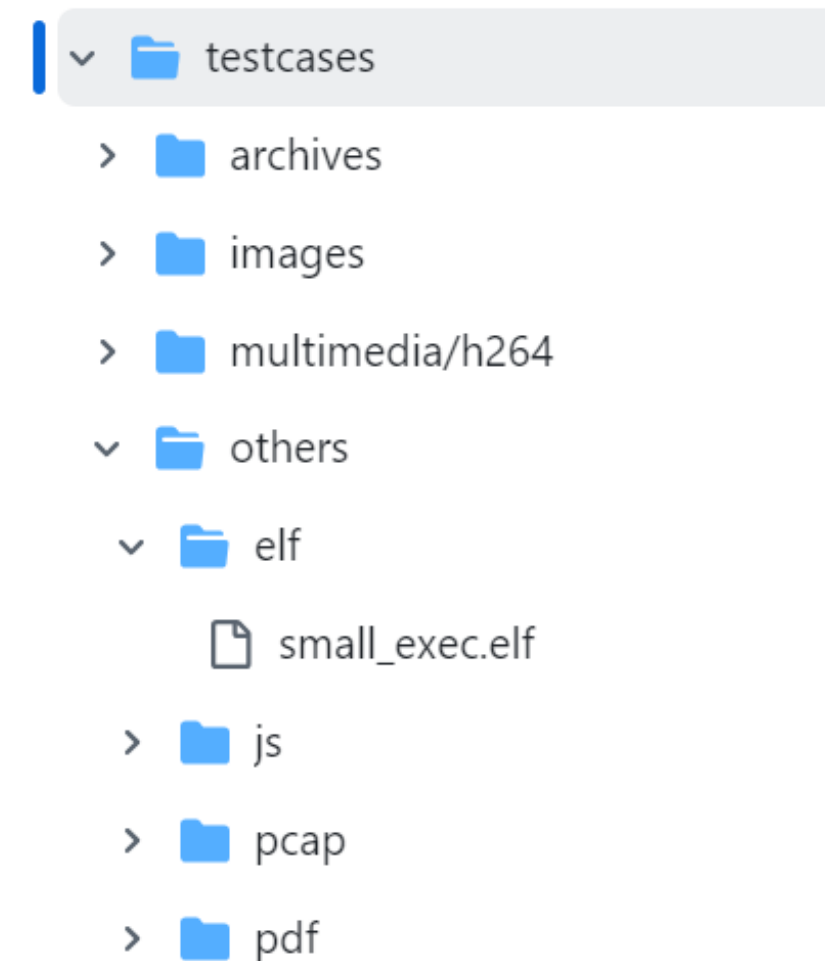
```
158
159     {
160         ZZIP_FILE* fp;
161         char      buf[17];
162         const char* name = argv[1] ? argv[1] : "README";
163
164         printf("Opening file '%s' in zip archive... ", name);
165         fp = zzip_file_open(dir, (char*) name, ZZIP_CASEINSENSITIVE);
166
167         if (! fp) {
168             printf("error %d: %s\n", zzip_error(dir), zzip_strerror_of(dir));
169         }
```

Calling a library function

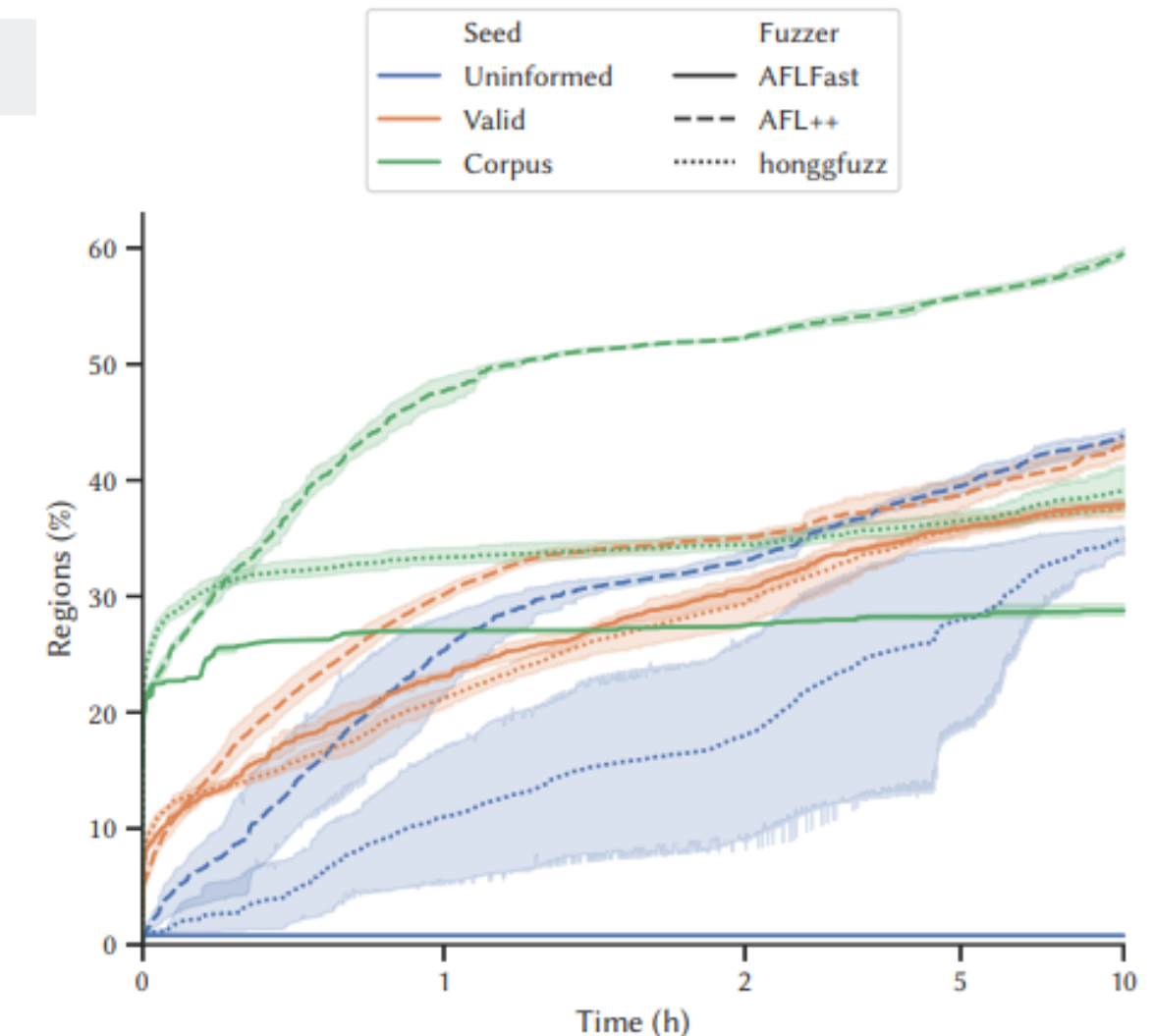
An example of a harness calling `zzip_file_open` of `libzzip`<sup>\*1</sup>

# Automation Blocker: #4 Initial Seed Files

- The efficiency of fuzzing, i.e., coverage, varies significantly depending on the presence or absence of initial seeds.
- Typically, expected input is manually understood.
  - Reading the README or manpage.
  - Search the Internet to find appropriate seeds.
    - AFL++ Testcases<sup>\*2</sup>
    - fuzzing-seeds<sup>\*3</sup>



AFL++ Testcases: Sample files in various formats are provided.



Fuzzing coverage comparison results for readelf.<sup>\*1</sup> : When an appropriate set of seeds (corpus: a set of seed files) was prepared, AFL++ showed approximately 1.5 times the difference in results.

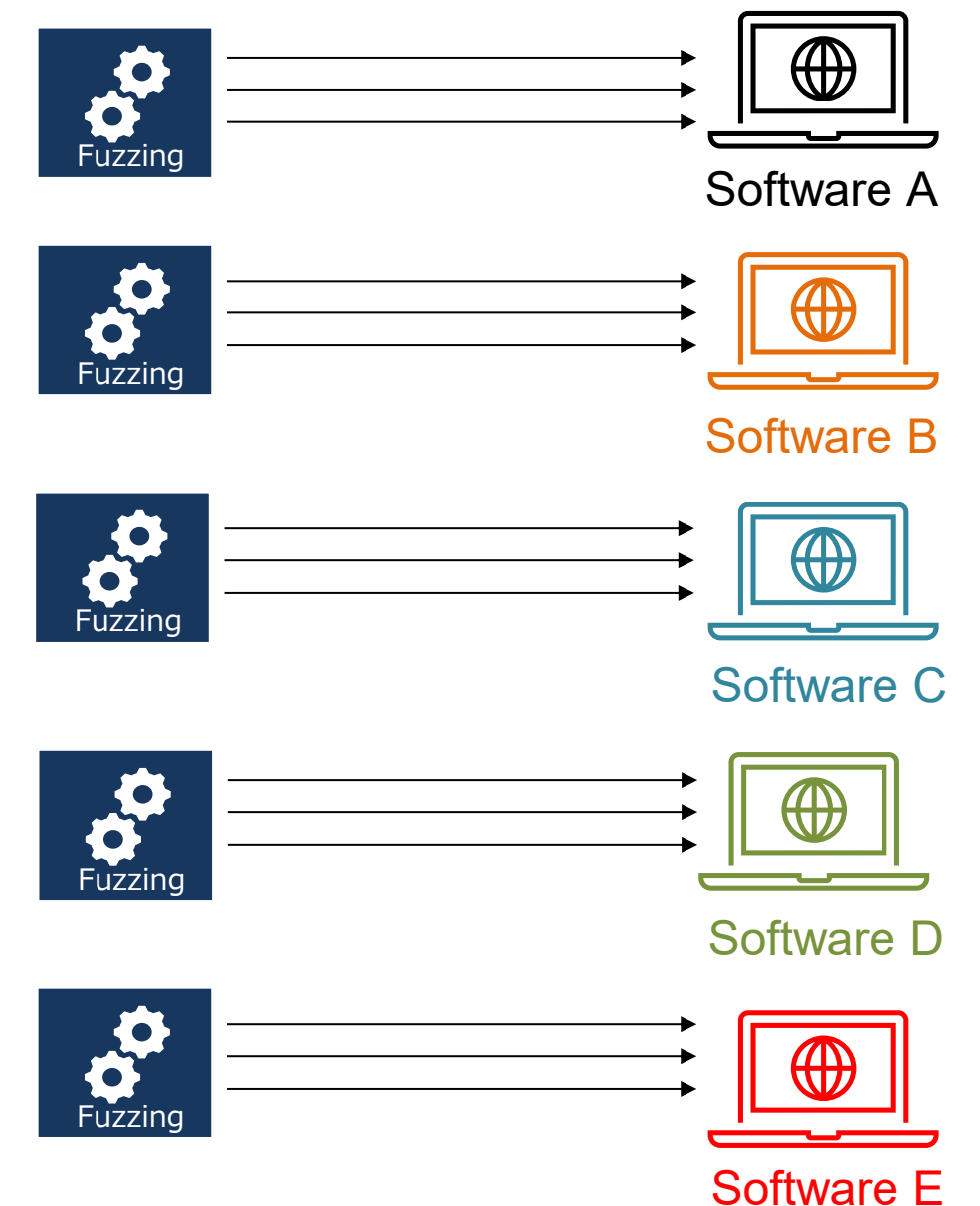
(\*1) "Seed selection for successful fuzzing", Adrian Herrera et al. <https://dl.acm.org/doi/10.1145/3460319.3464795>

(\*2) <https://github.com/AFLplusplus/AFLplusplus/tree/stable/testcases>

(\*3) <https://github.com/FuturesLab/fuzzing-seeds>

# Automation Blocker for Fuzzing

- Fuzzing does not necessarily automate the entire workflow of it.
  - It's necessary for people to properly prepare in advance.
- Tasks where human intervention is needed to (efficiently) perform fuzzing.
  - **#1 Harness Preparation**
  - #2 Command-line Arguments Analysis
  - #3 Building Binaries for Fuzzing
  - **#4 Initial Seed Files**
- How OSS-Fuzz deal with them?
  - Each OSS project individually prepare them.
- If solved, what would make you happy ?
  - We can bring the benefits and impact of fuzzing to OSS projects that cannot allocate sufficient resources for security testing (fuzzing tests), primarily smaller OSS projects.



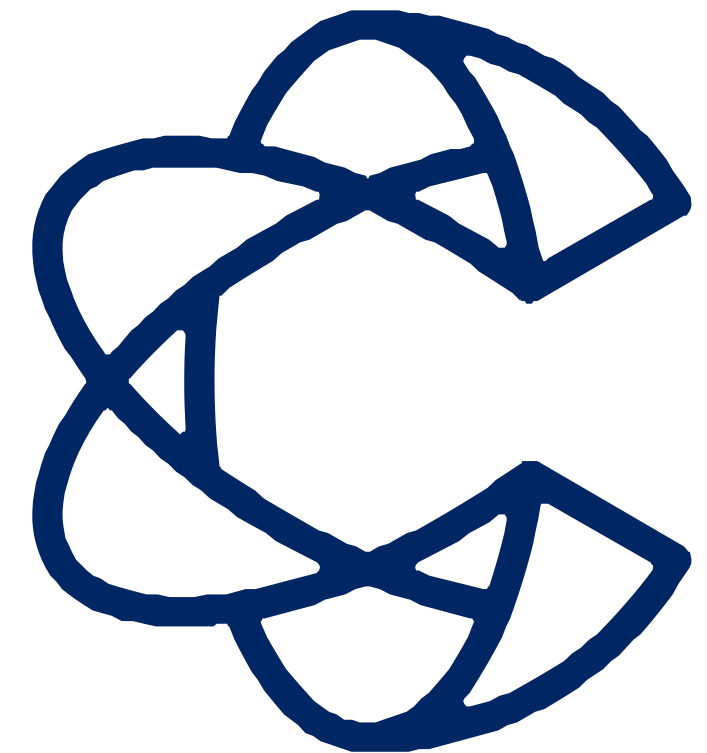
When continuously performing fuzzing on different software, human intervention decreases its efficiency.

# Agenda

- Vulnerability and Fuzzing

- **PkgFuzz**

- PkgFuzz Project
- From Crash to Exploit
- Next Step
- Wrap up





# Core Concept

- Monitor the build process of a software package (such as .deb) and automatically collect information needed for fuzzing.
- Advantages
  - As a test for the package, test code that calls library functions is included.
    - "Harness Preparation"
  - By monitoring the package tests, the appropriate command-line options and input files can be identified.
    - "Command-line Arguments" and "Initial Seed Files"
  - The method for passing build options is standardized, making it easy to add options for fuzzing.
    - "Building Binaries for Fuzzing"

```
$ apt source time (snip)
$ cd time-1.9/
$ debian/rules build (snip)
dh build
  dh_update_autotools_config
  dh_autoreconf
(snip)
gcc -DHAVE_CONFIG_H -I. (snip) -c -o src/time-time.o `test
-f 'src/time.c' || echo './'`src/time.c
(snip)
make check-TESTS (snip)
PASS: tests/help-version.sh
PASS: tests/time-max-rss.sh
(snip)
$ CC=clang debian/rules build (snip)
clang -DHAVE_CONFIG_H -I. (snip) -c -o src/time-time.o
`test -f 'src/time.c' || echo './'`src/time.c
(snip)
```

Package Source Download

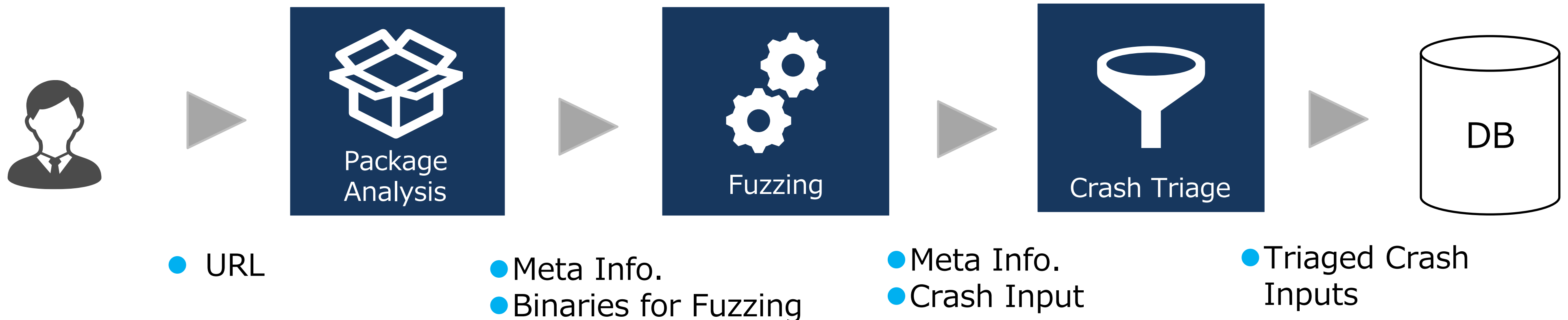
Source code build

Tests in building process

Customizable Build with Environment Variables

# PkgFuzz

- A system that implements the core concept to automate the entire fuzzing workflow.
  - 3 Phases: Package Analysis, Fuzzing, and Crash Triage<sup>\*1</sup>
  - By just providing the package URL, triaged crash inputs are accumulated in the DB.

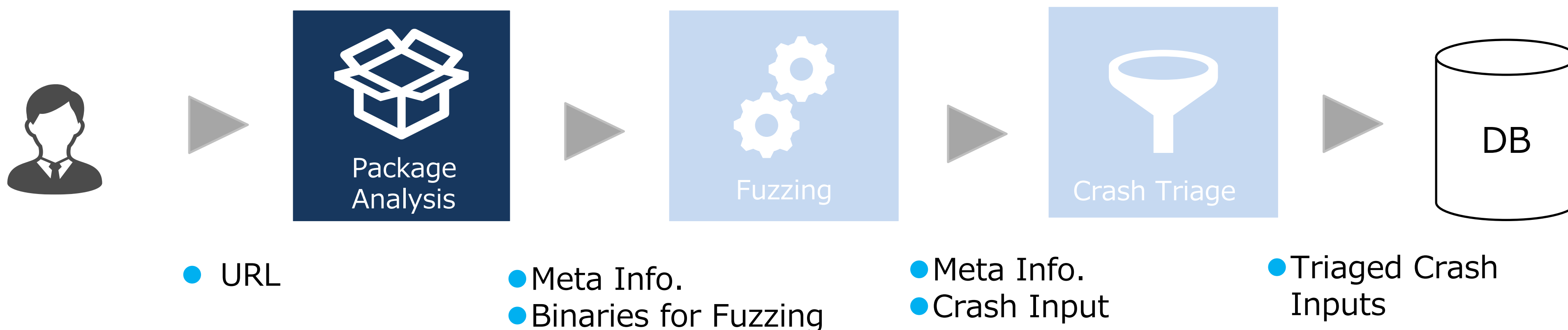


(\*1) Crash triage: categorizing a large number of crashes to select targets that should be prioritized for further analysis.

# Phase 1: Package Analysis

- 1 Download and build the software package (source package).
- 2 Monitor the build process and collect the following information.
  - Executed files.
  - Command-line arguments
  - Input files for tests
- 3 Specify the fuzzing compile options in the package build options and generate binaries for fuzzing.

Tools: modified execsnoop (A tool that hooks exec syscall family using BPF/eBPF.)



# Example: jbigkit v2.1-6

1 Observe the build process with execsnoop and extract logs where ELF executions are detected.

Time	Cmd	PID	PPID	ret of execve	Command-line arguments
02:10:42	tstcodec	3440740	3440738	0	"/.tstcodec"
02:10:45	tstcodec85	3442898	3440738	0	"/.tstcodec85"
02:10:46	jbgtopbm	3443661	3443655	0	"/.jbgtopbm" "../examples/ccitt1.jbg" "test-ccitt1.pbm"
02:10:46	jbgtopbm	3443723	3443666	0	"/.jbgtopbm" "test-t82-koXoyx5U.jbg85" "test-t82-koXoyx5U.pbm85"
02:10:46	pbmtojbg	3443686	3443655	0	"/.pbmtojbg" "test-ccitt1.pbm" "test-ccitt1.jbg"
02:10:46	pbmtojbg85	3443672	3443666	0	"/.pbmtojbg85" "-p" "0" "test-t82.pbm" "test-t82-koXoyx5U.jbg85"

2 Extract ones that take a file as an argument.

Time	Cmd	PID	PPID	ret of execve	Command-line arguments
02:10:42	tstcodec	3440740	3440738	0	"/.tstcodec"
02:10:45	tstcodec85	3442898	3440738	0	"/.tstcodec85"
02:10:46	jbgtopbm	3443661	3443655	0	"/.jbgtopbm" "../examples/ccitt1.jbg" "test-ccitt1.pbm"
02:10:46	jbgtopbm	3443723	3443666	0	"/.jbgtopbm" "test-t82-koXoyx5U.jbg85" "test-t82-koXoyx5U.pbm85"
02:10:46	pbmtojbg	3443686	3443655	0	"/.pbmtojbg" "test-ccitt1.pbm" "test-ccitt1.jbg"
02:10:46	pbmtojbg85	3443672	3443666	0	"/.pbmtojbg85" "-p" "0" "test-t82.pbm" "test-t82-koXoyx5U.jbg85"

3 The final command line used for fuzzing.

```
afl-fuzz -T jbigkit -i ./in -o ./out jbigkit-2.1/./debian/tmp/usr/bin/jbgtopbm @@ test-ccitt1.pbm
```

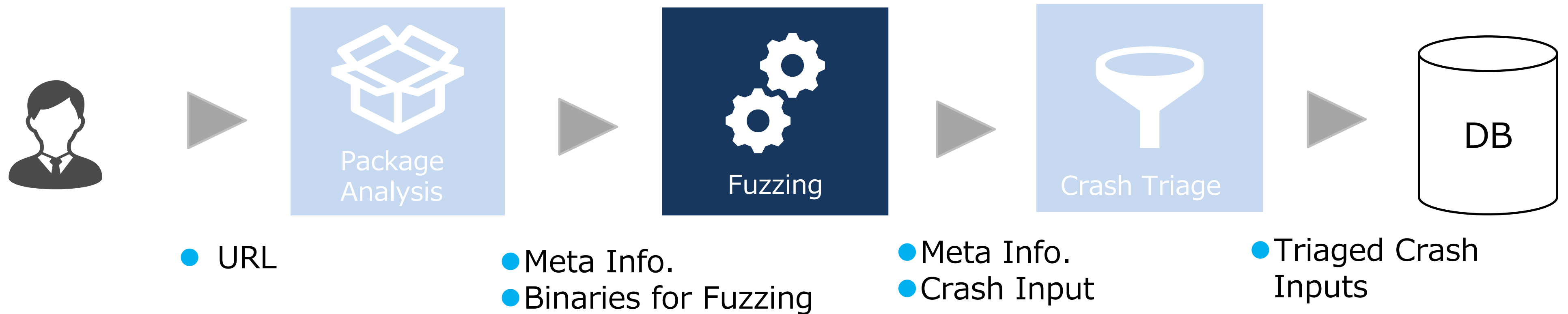
Complete the command path.

Identify the input file in the command line (replace it with "<<" to notify the fuzzer of its position).

# Phase 2: Fuzzing

- 1 Short-term fuzzing
  - Check for errors and the coverage growth (i.e., corpus expansion status).
- 2 After the above checks, perform long-term fuzzing.

Tools: AFL++ 4.08c (Fuzzer) + Address Sanitizer (Sanitizer)

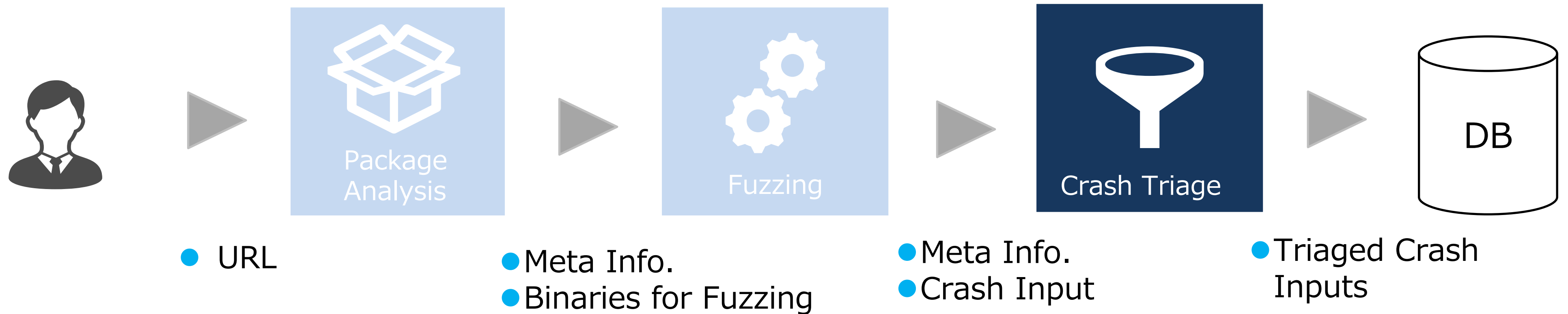




# Phase 3: Crash Triage

- 1 Classify and eliminate duplicated crashes.
- 2 Simply assess exploitability and prioritize accordingly.
- 3 Verify reproducibility with the original binary.

Tools: CASR (Crash Analysis and Severity Report) + AFLTriage



# CASR (Crash Analysis and Severity Report)

- A tool for collecting crash reports, triaging, and assessing severity.
  - Its evaluation logic is based on GDB's exploitable plugin.
- Crash Type
  - Determined based on signal information and sanitizer output.
    - AbortSignal, stack-buffer-overflow(write), etc
- Severity
  - Classified into three levels based on type.
    - High (**EXPLOITABLE**): SegFaultOnPc, Write BoF, etc
      - A crash considered likely to allow easy control takeover.
    - Middle (**PROBABLY\_EXPLOITABLE**): BadInstruction, BoF except for the above.
      - A crash likely to be assessed for control takeover with some additional analysis.
    - Low (**NOT\_EXPLOITABLE**): AbortSignal, double-free, Read BoF, etc
      - A crash unlikely to allow control takeover.

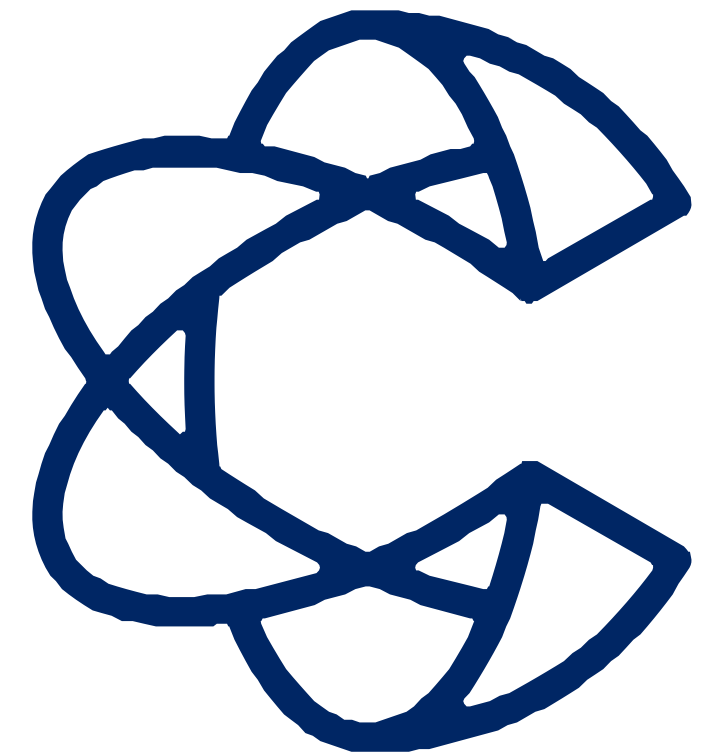
Ref: <https://github.com/ispras/casr/blob/master/docs/classes.md>

# Agenda

- Vulnerability and Fuzzing
- PkgFuzz

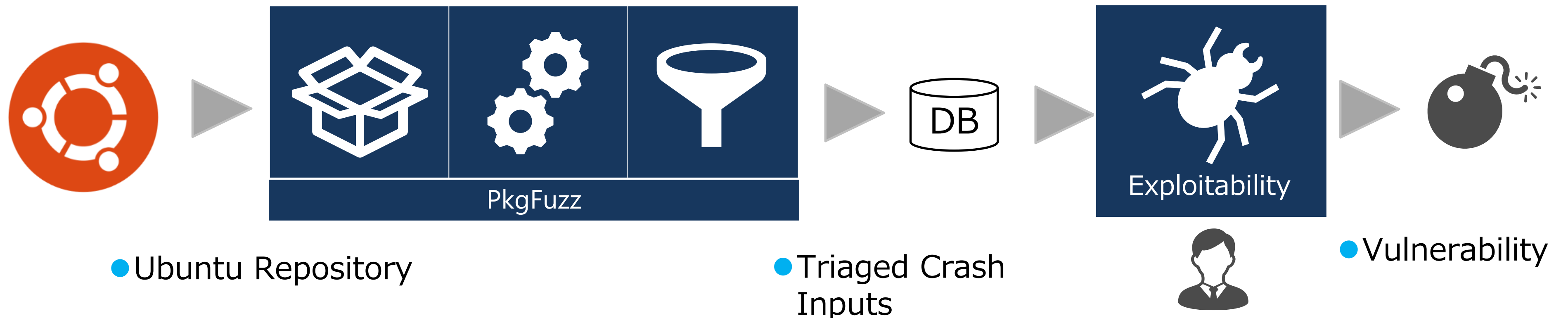
## ● PkgFuzz Project

- From Crash to Exploit
- Next Step
- Wrap up



# PkgFuzz Project

- **Objective** : Vulnerability investigation of Ubuntu packages and performance evaluation of PkgFuzz.
- **Target** : 265 Debian packages from Ubuntu 23.10.
- **Approach** : Fuzzing the target using PkgFuzz and manual exploitability verification.
  - PkgFuzz automatically generates crash inputs.
  - A human examines PkgFuzz's output and creates exploit code for verification.



# Package Selection

Filter Conditions		# of Packages
1	Total number of Ubuntu packages	71,858
2	Number of packages with available data	71,277
3	Number excluding packages from the exclusion list <sup>*1</sup>	46,585
4	Number aggregated by source packages	24,135
5	Packages with over 1K lines of C/C++ code <sup>*2</sup>	10,116
6	First entry in ChangeLog is before 2015/01/01 <sup>*3</sup>	6,915
7	Packages that remained through the long-term fuzzing selection process by PkgFuzz <sup>*4</sup>	<b>265</b>

(\*1): Exclude items that are difficult to handle as fuzzing targets, such as the Linux kernel or language processors.  
(:2): Projects with a small amount of code are less likely to contain bugs, so only those with 1K lines or more are targeted.  
(\*3): New projects are likely to use the latest software engineering practices and tools, making them less likely to contain bugs, so those with older code are targeted.  
(\*4): Exclude packages that fail to build in the package analysis phase or those that do not achieve coverage growth during the short-term fuzzing phase.



# Overview of PkgFuzz Project Results

■ The following results were obtained from fuzzing with PkgFuzz over 318 days (2023/9/19 ~ 2024/7/24).

Item	Results
Long-term fuzzing target packages	265 Packages
Total number of fuzzing executions	606 (2.28 / package <sup>*1</sup> )
Total number of crashes	64,658
Total number of triaged crashes	1,186 (54.5 crashes / bug <sup>*2</sup> )

(\*1) If different command-line arguments are observed for the same package, fuzzing is attempted multiple times.

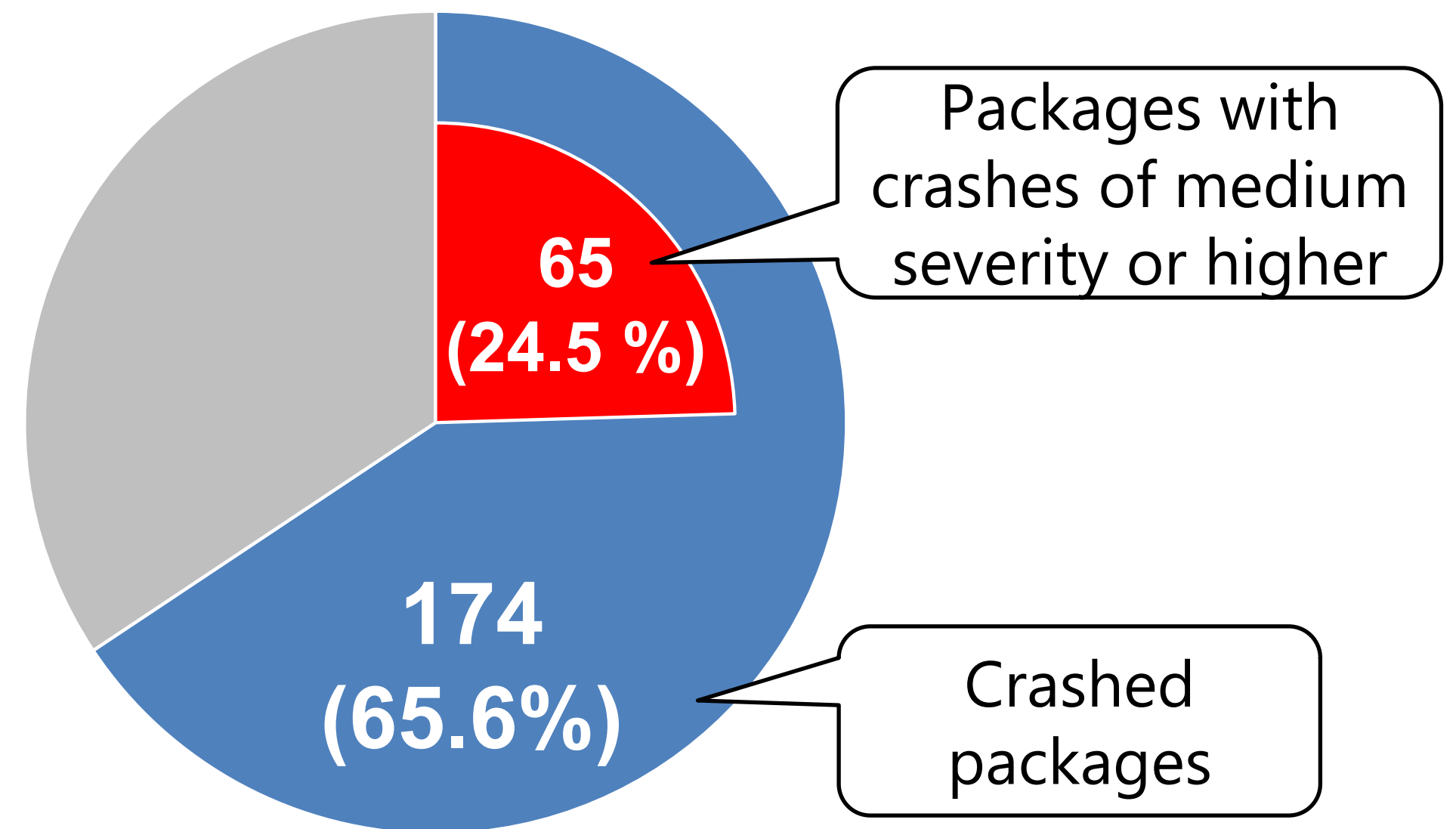
(\*2) AFL++ is likely to generate inputs similar to the crashing ones, as it adds inputs that expand coverage to the seed.

# Result Analysis

- Based on the above results, analyze the findings from the following perspectives:
  - Proportion of packages containing vulnerabilities
    - Proportion of packages where crashes were observed
    - Causes of crashes
  - Efficiency of fuzzing with PkgFuzz
    - Comparison with human-involved fuzzing (OSS-Fuzz)
    - Characteristics of packages where no crashes were observed
    - Appropriate fuzzing execution time

# Proportion of packages containing vulnerabilities

- Proportion of packages where crashes were observed
  - The number of packages with at least one crash is **174 (=65.6%)**.
  - **Approx. 2/3 of the packages contain bugs.**
- Proportion of packages likely to contain vulnerabilities
  - The number of packages with one or more crashes of medium severity or higher is **65 (=24.5%)**.
  - **Approx. 1/4 of the packages are likely to contain vulnerabilities.**

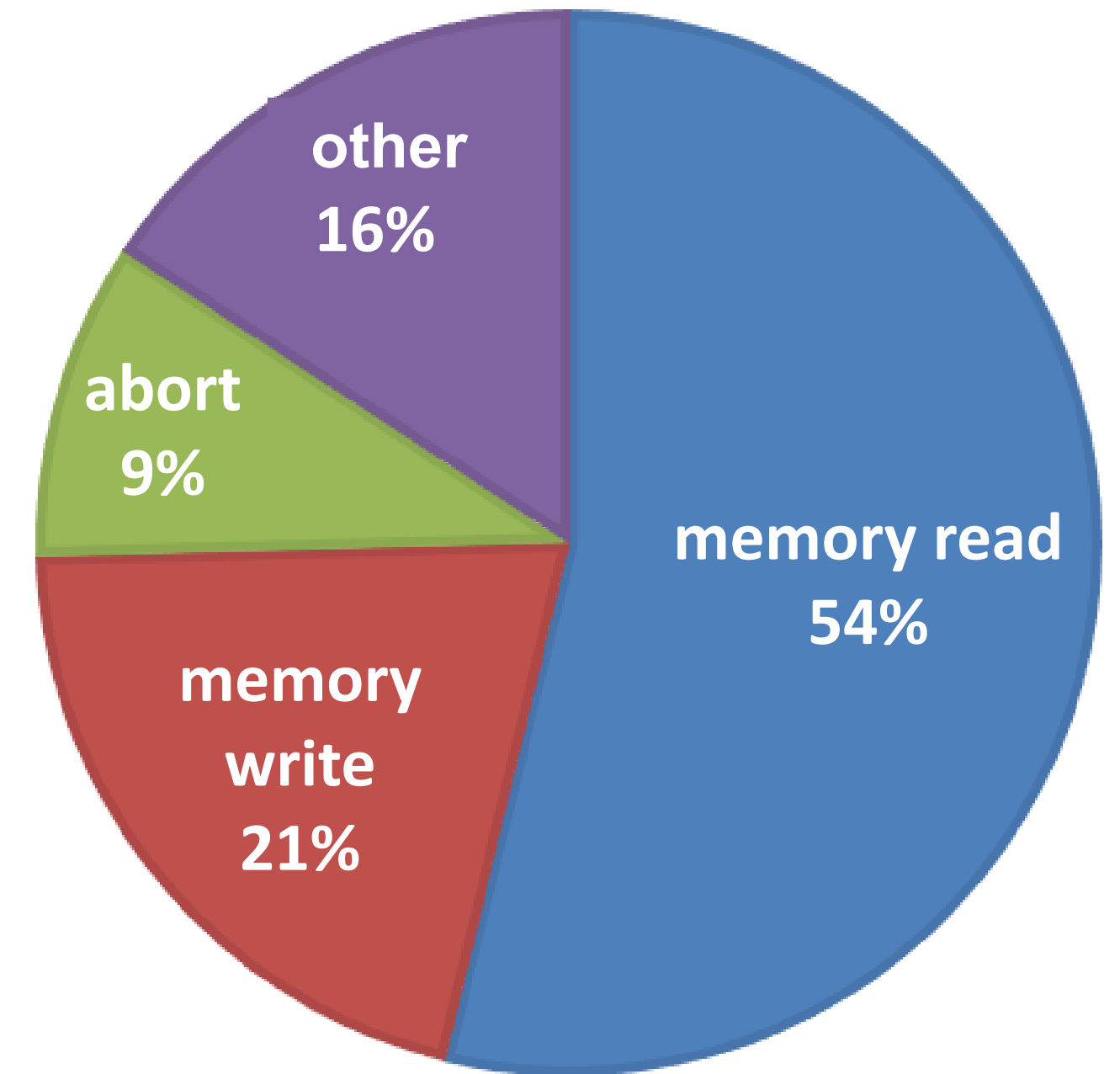


# Proportion of packages containing vulnerabilities

## ■ Causes of Crashes

→ According to CASR, the primary cause is memory reads and writes.

- Memory Read: **1,123 crashes (54% of total)**
  - Memory reads contribute to crashes, but determining whether it is limited only to reading would typically require further investigation<sup>\*1</sup>.
- Memory Write: **435 crashes (21% of total)**
  - Due to its direct link to vulnerabilities, it is rated as highly severe.



(\*1): ASanity: On Bug Shadowing by Early ASan Exits

# Efficiency of fuzzing with PkgFuzz

- Comparison with Human-Involved Fuzzing (OSS-Fuzz)
  - # of common OSS (packages) between PkgFuzz Project and OSS-Fuzz: 32 (about 12%)
  - Comparison based on code coverage and # of bugs detected by fuzzing
- Results
  - Line Code Coverage
    - # of common OSS for which line coverage could be measured: 18
    - **There is an average difference of 14 points, but in some cases, PkgFuzz achieved higher coverage.**
  - # of Bugs Detected
    - **OSS-Fuzz detected more bugs per OSS on average, but there was no significant difference in the vulnerability rate.**

	Averaged Line Coverage	# of Bugs Detected		Vulnerability Rate
		Total	Par Package	
OSS-Fuzz	54.65%	1000>	21.2	26%
PkgFuzz	40.80%	1186	4.47	18%

Package	OSS-Fuzz	PkgFuzz
libpsl	19.22%	51.01%
speex	35.09%	43.56%
ninja	36.09%	39.26%
...	...	...

Ref. OSS-Fuzz: Five months later, and rewarding projects | Google Open Source Blog, <https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html>



# Efficiency of fuzzing with PkgFuzz

## ■ Pattern of Packages with No Observed Crashes

- Coverage reaches saturation early
  - e.g., w3m

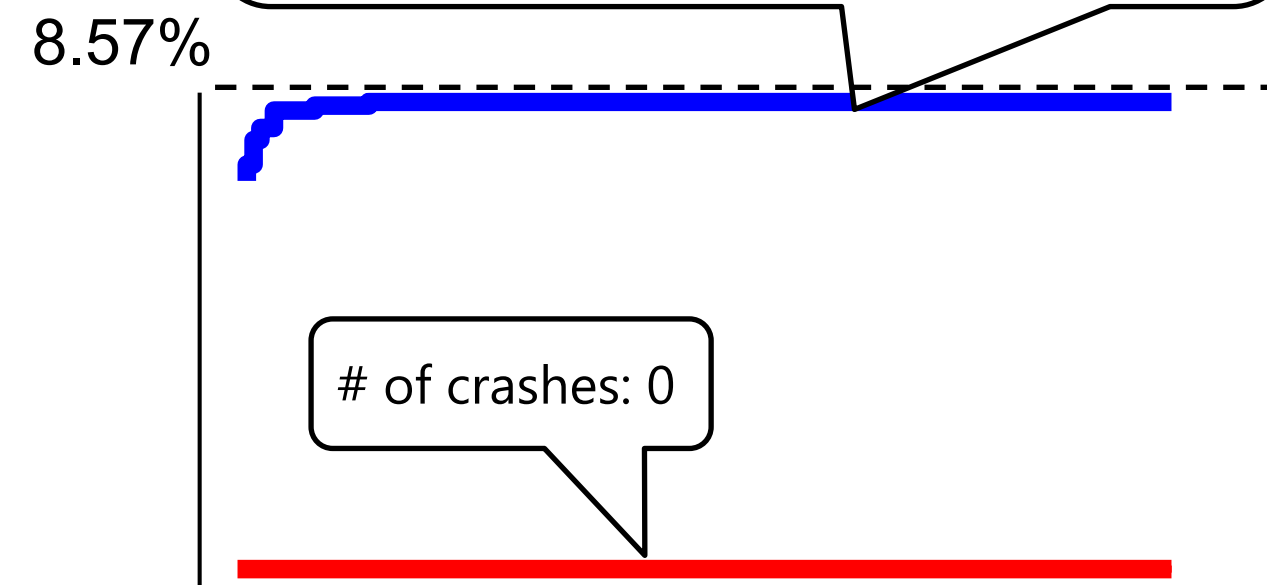
## ■ Cause Analysis

- Fuzzing blockers, such as magic numbers, checksums, and keywords
  - Analyzed with Fuzz Introspector

## ■ Improvement

- Combine different types of fuzzers in an ensemble approach (e.g., AFL++, libFuzzer, Honggfuzz), or employ hybrid fuzzing techniques that integrate program analysis methods.

Fuzzing was conducted for approximately 6 days, with almost no change in the bitmap\_cvg (edge coverage) value, ending at 8.57%.



```
1823         if (c == EOF)
1824             unexpected_EOF();
1825         if (c != '%')
1826             syntax_error(lineno, line, cptr);
1827         switch (k = keyword())
===== FUZZ BLOCKER DETECTED BY FUZZ INTROSPECTOR =====
1828         {
1829             case MARK:
1830                 return;
1831
1832             case IDENT:
===== BLOCKED
1833                 copy_ident();
1834                 break;
1835
1836             case XCODE:
===== BLOCKED
1837                 copy_code();
1838                 break;
1839
1840             case TEXT:
===== REACHED
1841                 copy_text();
1842                 break;
1843
1844             case UNION:
===== BLOCKED
1845                 copy_union();
1846                 break;
```

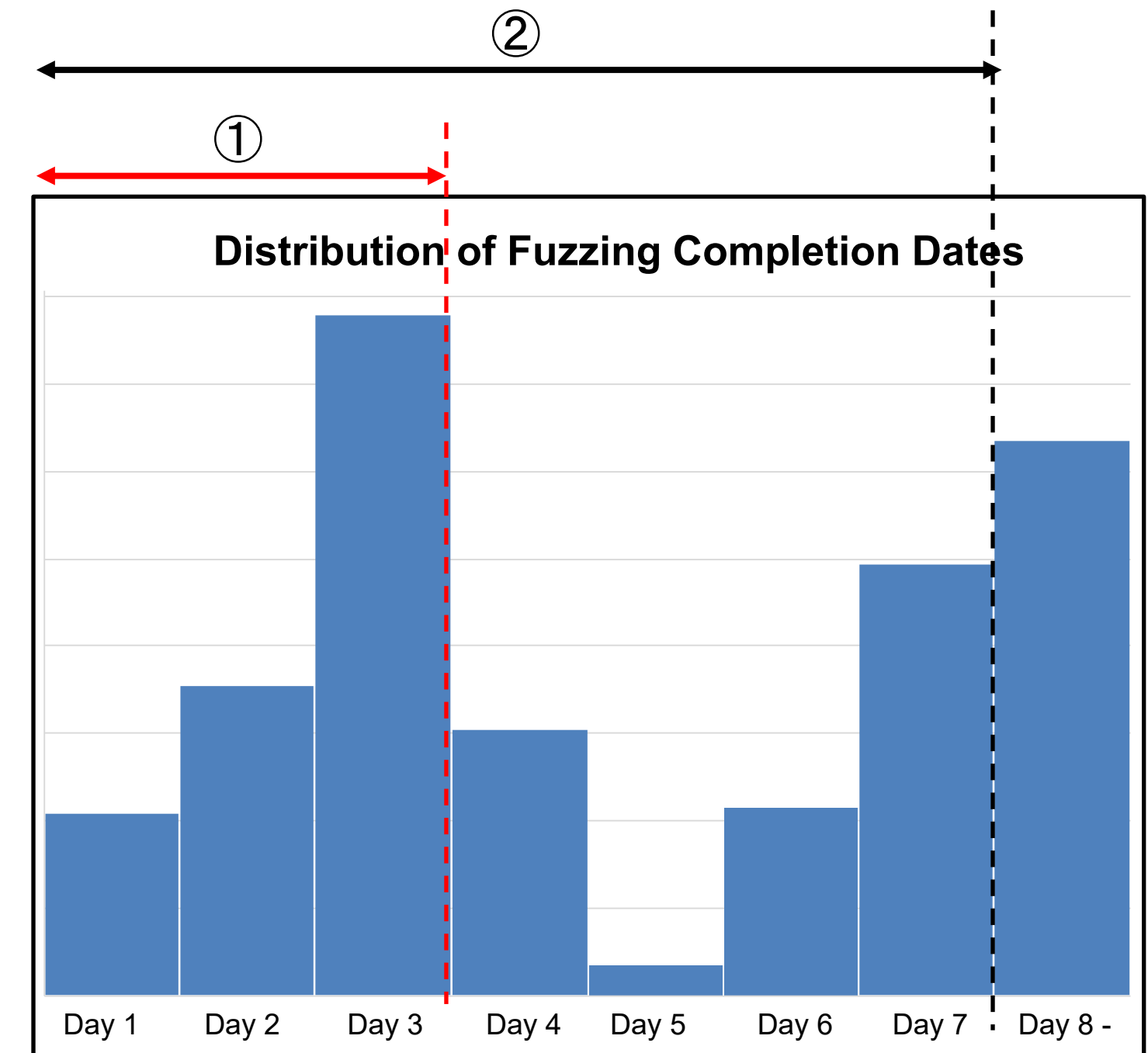
# Efficiency of fuzzing with PkgFuzz

## ■ Optimal Fuzzing Duration

- 44% of total fuzzing completes within 3 days (①), and 79% completes within 1 week (②).
- Termination criteria for fuzzing in the PkgFuzz Project
  - 1 week (based on experience) + human judgment (coverage expansion rate and # of parallel instances)
  - Given the observed trend of coverage increase during fuzzing<sup>\*1</sup>, the current criteria may not be optimal.

## ■ Improvement

- By Referencing existing research<sup>\*2\*3</sup>, consider identifying critical areas to inspect in advance, then guiding fuzzing specifically for those areas.



(\*1) It is said to exhibit behavior where "instead of gradually increasing over time, it suddenly spikes and then remains nearly constant for a certain period," repeating this pattern. (punctuated equilibrium)

(\*2): Green Fuzzing: A Saturation-based Stopping Criterion using Vulnerability Prediction, <https://mboehme.github.io/paper/ISSTA23.pdf>

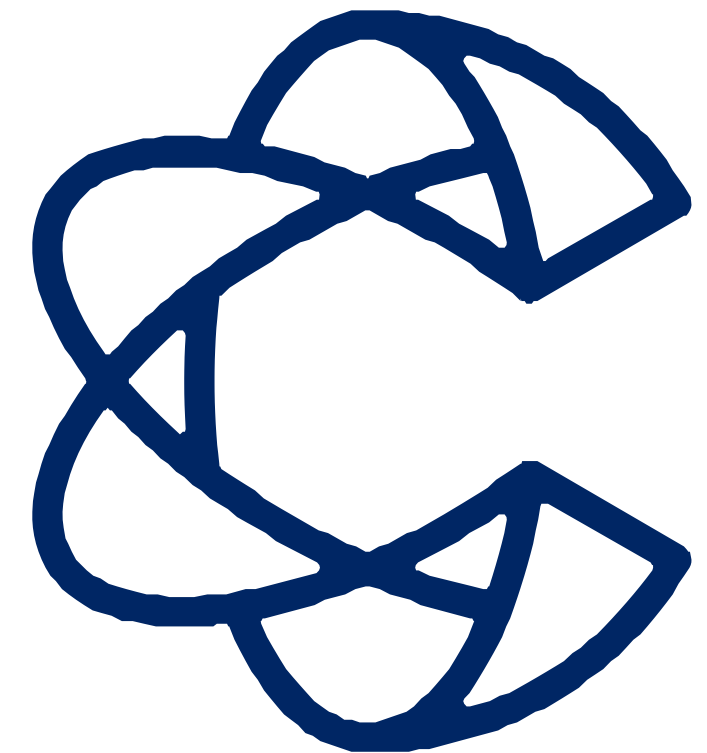
(\*3): SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing

# Summary of Result Analysis

- Proportion of packages containing vulnerabilities
  - Approx. 1/4 packages (65 out of 265) potentially contain vulnerabilities. However, crashes due to memory reads may underestimate the severity of vulnerabilities.
- Efficiency of Fuzzing with PkgFuzz
  - Compared to OSS-Fuzz, the PkgFuzz Project was able to perform fuzzing automatically (without human involvement) with results comparable to human-involved fuzzing.
  - Automation allowed fuzzing to be conducted on each OSS with low resource requirements. This enabled fuzzing of OSS that OSS-Fuzz does not target, detecting many additional bugs (only a 12% overlap with OSS-Fuzz = 32 out of 265).
  - There is room for improvement in avoiding fuzzing blockers and configuring termination criteria. By incorporating findings from existing research into the PkgFuzz Project, a more efficient fuzzing system can be designed.

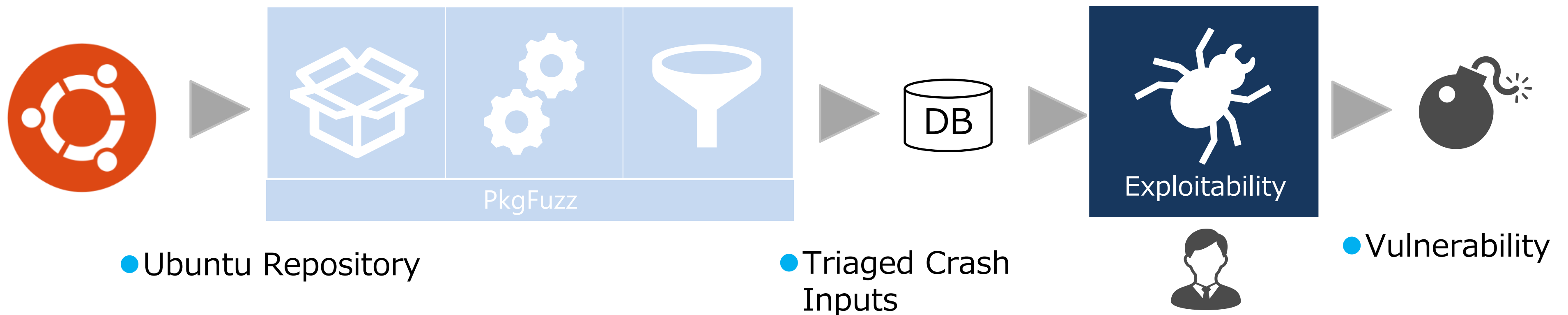
# Agenda

- Vulnerability and Fuzzing
- PkgFuzz
- PkgFuzz Project
- **From Crash to Exploit**
- Next Step
- Wrap up



# Exploitability Verification

- Even if a crash is automatically classified as (probably) exploitable by triage tools, it is still just an assumption, so further detailed verification is necessary.
- For 65 packages (with 484 crash inputs) flagged as (probably) exploitable, a manual verification was conducted to determine if they were truly exploitable.
  - Step1: Use Case Analysis (65 pkgs→22 pkgs)
  - Step2: Exploitability Analysis (22 pkgs→4 pkgs)





# Step1: Use Case Analysis

- **Objective** : Investigate the program's use cases when a crash occurs and filter down the target based on the feasibility of exploitation.
- **Method** : Manually investigate use cases from package documentation, excluding cases where the program is used in situations where an attacker cannot control the input from outside, as attacking would be difficult even if exploitable.
  - Exclusion Examples:
    - Tools used as part of the build process
    - Parser for configuration files of server software
- **Result** : Narrowed down from 65 packages to 22 by excluding unrealistic use cases.

# Step2 : Exploitability Analysis

- **Objective:** Demonstrate that the crash is exploitable by creating exploit code that can actually take control.
  - Decisions by the triage tool (CASR) are only considered as reference information.
- **Method:** Through detailed source code investigation and crash analysis using a debugger, the root cause of the bug was identified, and attempt was made to manually create the exploit code.
- **Result:** Exploit code was successfully created for 4 out of the 22 packages (5 crashes).
  - Vulnerability breakdown: Stack Buffer Overflow: 3, Heap Buffer Overflow: 2
  - Other cases: Difficult to use as vulnerability, software specification, etc.

# Example : Assimp(CVE-2024-40724)

## ■ Open Asset Import Library

- A library for converting various 3D file formats into formats that can be handled by a program.

## ■ Attack Scenario

- An attacker tricks users of a program that loads 3D model data using Assimp into opening a manipulated PLY file (via internet distribution, SE attacks, etc.)

## ■ Vulnerability Cause

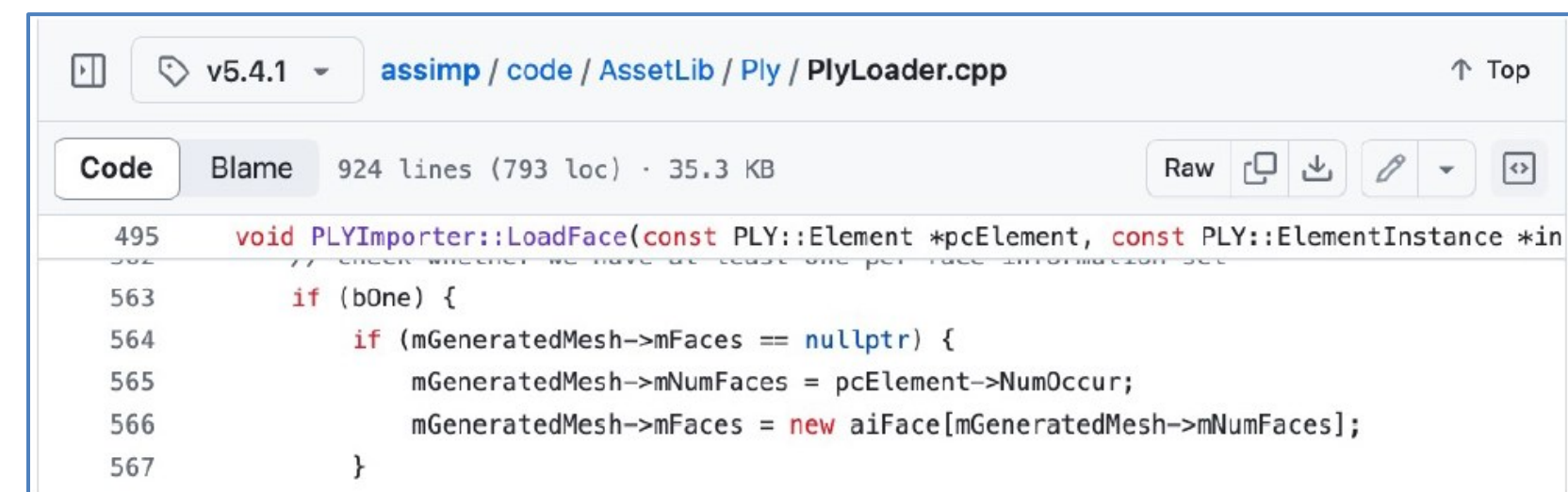
- In the PLY file loading code of Assimp, there is a flaw in memory allocation , which makes a **Heap Overflow** possible when reading certain elements multiple times

## ■ Exploitation

- By causing a heap overflow and corrupting adjacent metadata, arbitrary address write becomes possible, allowing the attacker to overwrite pointers and gain shell access.

```
ply
format binary_
element vertex 8
property float x
element face 30
property list int8 int
vertex_index
element face 31
property list char int
...
```

Example PLY file



```
495 void PLYImporter::LoadFace(const PLY::Element *pcElement, const PLY::ElementInstance *in
563 if (bOne) {
564     if (mGeneratedMesh->mFaces == nullptr) {
565         mGeneratedMesh->mNumFaces = pcElement->NumOccur;
566         mGeneratedMesh->mFaces = new aiFace[mGeneratedMesh->mNumFaces];
567     }
```

Location of vulnerability cause

```
root@427467773baf:~# ./assimp/bin/assimpd info payload
Launching asset import ... OK
Validating postprocessing flags ... OK
# id
uid=0(root) gid=0(root) groups=0(root)
# whoami
root
#
```

Execution of exploit code

# Vulnerability Report

- We reported the 5 vulnerabilities that allow arbitrary code execution to the developers via IPA and JPCERT/CC.

Package	Reported	Accepted	Developer Notification	Patch Release	Advisory	CVE	CVSS
sdop	3/4	6/6	6/13	6/25	7/29	CVE-2024-41881	7.0
orc	4/17	6/11	7/9	7/19	7/26	CVE-2024-40897	7.0
assimp(1)	5/21	6/11	7/1	7/7	7/18	CVE-2024-40724	8.4
assimp(2)	6/19	7/10	7/22	8/31	9/18	CVE-2024-45679	8.4
xfpt	8/5	8/6	8/7	8/12	8/29	CVE-2024-43700	7.0

# Advisory (xfpt)

CVE

CVE-2024-43700

PUBLISHED

View JSON

User Guide

Collapse all

Required CVE Record Information

CNA: JPCERT/CC

Published: 2024-08-29

Updated: 2024-08-29

Description

xfpt versions prior to 1.01 fails to handle appropriately some parameters inside the input data, resulting in a stack-based buffer overflow vulnerability. When a user of the affected product is tricked to process a specially crafted file, arbitrary code may be executed on the user's environment.

Product Status

Learn more

Vendor

Philip Hazel

Product

xfpt

Versions1 Total

Default Status: unknown

Affected

affected at prior to 1.01

Published:2024/08/28Last Updated:2024/08/28

JVNVU#96498690

xfpt vulnerable to stack-based buffer overflow

Overview

xfpt contains a stack-based buffer overflow vulnerability.

Products Affected

xfpt versions prior to 1.01

Description

xfpt fails to handle appropriately some parameters inside the input data, resulting in a stack-based buffer overflow vulnerability (CWE-121).

Impact

When a user of the affected product is tricked to process a specially crafted file, arbitrary code may be executed on the user's environment.

Vulnerability Analysis by JPCERT/CC

CVSS v3

CVSS:3.0/AV:L/AC:H/PR:N/UI:R/S:U/C:H/I:H/A:H

Base Score: 7.0

Comment

AC(Attack Complexity) is evaluated as High considering that exploit protection mechanisms such as ASLR and stack canaries become popular in major OS environments.

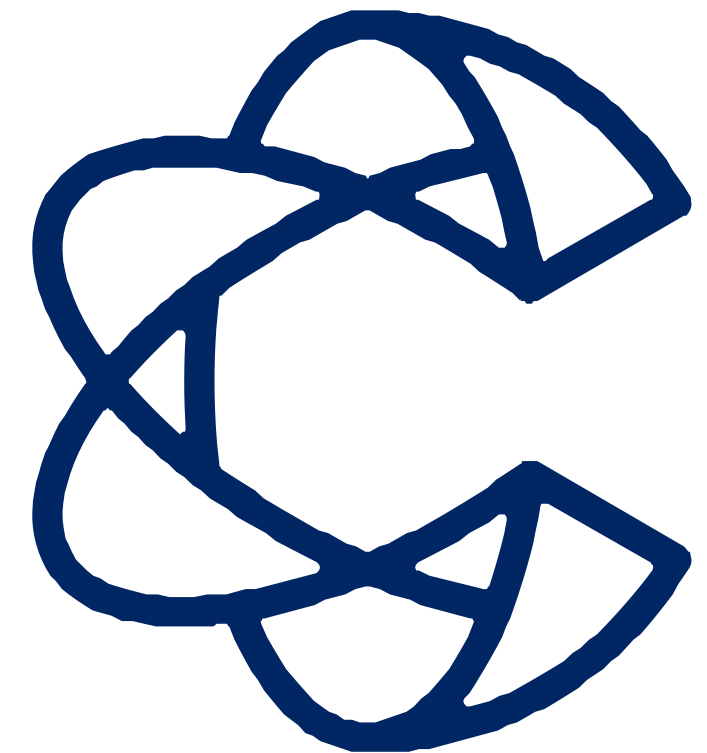
Credit

Yuhei Kawakoya of NTT Security Holdings Corporation reported this vulnerability to JPCERT/CC. JPCERT/CC coordinated with the developer.



# Agenda

- Vulnerability and Fuzzing
- PkgFuzz
- PkgFuzz Project
- From Crash to Exploit
- **Next Step**
- Wrap up





# Toward fully-automated system

- In the PkgFuzz Project, fuzzing was conducted on 265 packages, successfully overcoming some fuzzing automation blockers. However, many challenges were also identified in the process.
- Next, we will discuss challenges related to further automation:
  - Harness generation
  - Root cause analysis (RCA)

# Harness generation

- **Challenges in PkgFuzz:** Packages without tests or do not accept file paths as arguments during execution were excluded from fuzzing. Addressing these packages would require creating harnesses.
- **Existing Approaches:**
  - Harness creation is often done manually, though several methods and tools have been proposed and released for automation.
  - Approaches using LLMs are also promising, but they face challenges that need to be addressed (e.g., handling complex API specifications, cost of use). A hybrid approach combining LLMs and program analysis is likely necessary<sup>\*1</sup>.
- **Our Approach:** In the PkgFuzz Project, we aim to increase automation for a broader range of packages by adopting automated harness generation methods in the future.

(\*1) How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation (ISSTA'24)

# Root Cause Analysis (RCA)

- **Challenges in PkgFuzz:** One step in investigating exploitability involves identifying the root cause of crashes, a process currently done manually, which is time-consuming.
  - The root cause of a bug is not always the same as the crash location.
- **Existing Approaches:**
  - Typically, crash locations are investigated manually by reading program code with runtime information from the crash as a guide, and some automated tools and methods have also been proposed.
    - Statistical approach based on runtime differences between crash and non-crash inputs (e.g., Aurora, VulnLoc, etc.)
    - Rule-based approach based on symbolic execution (e.g., Arcus)
  - Practical use of these tools is still limited by issues with accuracy and usability.
- **Our Approach:** The PkgFuzz Project aims to automate root cause analysis by improving the accuracy and usability challenges of existing tools.

# Example of Analysis with RCA Tools

- Testing whether existing tools can identify the root cause of vulnerabilities discovered by PkgFuzz.
  - The tool outputs a ranked list of potential root cause locations (file name/line number).
  - The accuracy is checked by seeing where the manually identified correct cause ranks on the list (analysis time: 12 hours).

```
[INSN-0] 0x0805e323 -> read.c:95 (l2norm: 1.414214; normalized(N): 1.000000; normalized(S): 1.000000)
[INSN-1] 0x0805e413 -> read.c:119 (l2norm: 1.289673; normalized(N): 1.000000; normalized(S): 0.814405)
[INSN-2] 0x0805e306 -> read.c:93 (l2norm: 1.289673; normalized(N): 1.000000; normalized(S): 0.814405)
[INSN-3] 0x0805f4a8 -> read.c:603 (l2norm: 1.219202; normalized(N): 1.000000; normalized(S): 0.697463)
[INSN-4] 0x0805f48b -> read.c:599 (l2norm: 1.219202; normalized(N): 1.000000; normalized(S): 0.697463)
[INSN-5] 0x0805f44d -> read.c:596 (l2norm: 1.219202; normalized(N): 1.000000; normalized(S): 0.697463)
[INSN-6] 0x0805f3af -> read.c:585 (l2norm: 1.069146; normalized(N): 1.000000; normalized(S): 0.378250)
[INSN-7] 0x0805f3bd -> read.c:586 (l2norm: 1.067239; normalized(N): 1.000000; normalized(S): 0.372826)
[INSN-8] 0x0805f3a1 -> read.c:585 (l2norm: 1.067239; normalized(N): 1.000000; normalized(S): 0.372826)
[INSN-9] 0x0805f356 -> read.c:581 (l2norm: 1.050842; normalized(N): 1.000000; normalized(S): 0.322908)
...
```

Root Cause Candidates (sdop)

- **sdop**/CVE-2024-41881 [Success Case]
  - The tool accurately identified the root cause, pinpointing it at the top of the ranked list (rank 1).
- **assimp**/CVE-2024-40724 [Failure Case]
  - The tool failed to accurately identify the root cause (ranked below 300).
  - This failure is likely because the root cause location is encountered in both crash and non-crash states.

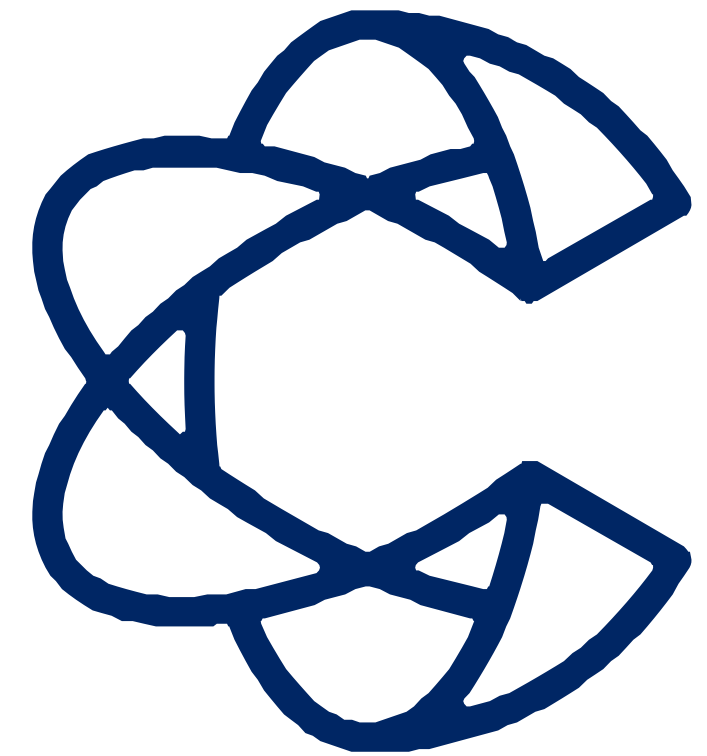


Actual Fix Patch (sdop)

- Useful when succeeds, but the burden on the analyst increases in the event of failure, so improvements in accuracy and usability are necessary for practical use.

# Agenda

- Vulnerability and Fuzzing
- PkgFuzz
- PkgFuzz Project
- From Crash to Exploit
- Next Step
- **Wrap up**

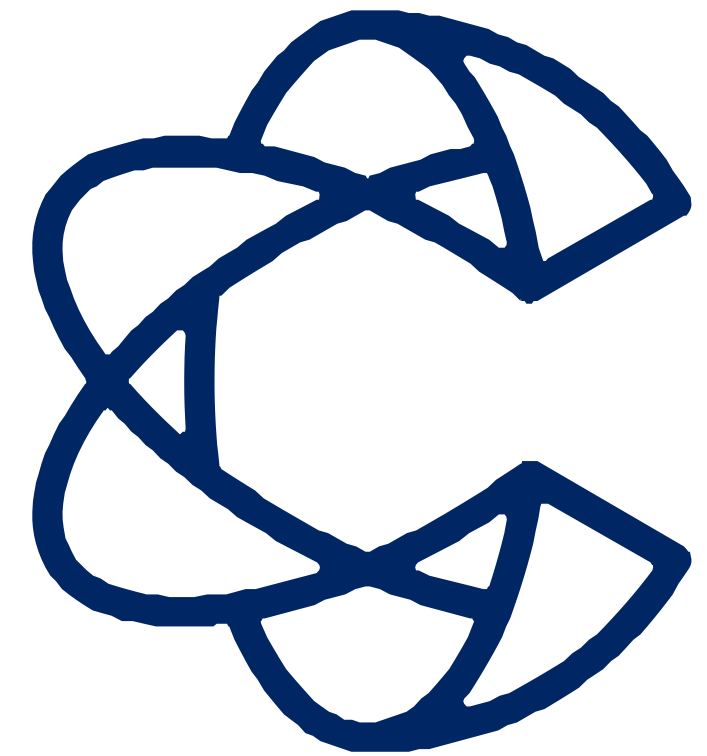


# Wrap up

- Introduced the PkgFuzz Project as another fuzzing project for OSS.
  - Through the automation provided by PkgFuzz, the project automatically detected 64,658 crashes (reduced to 1,186 after triage) across 265 Debian packages, leading to five advisories and CVE publications.
- Compared with OSS-Fuzz, the PkgFuzz Project achieved nearly equivalent results automatically (without human intervention) as would be expected with manual fuzzing. Moreover, PkgFuzz was able to conduct fuzz testing on OSS that OSS-Fuzz does not target (such as smaller OSS), with only a 12% overlap (32 out of 265 packages).
- We further analyzed existing research to explore possibilities for additional automation within the PkgFuzz Project.



# Appendix



# Types of Fuzzing

Simple

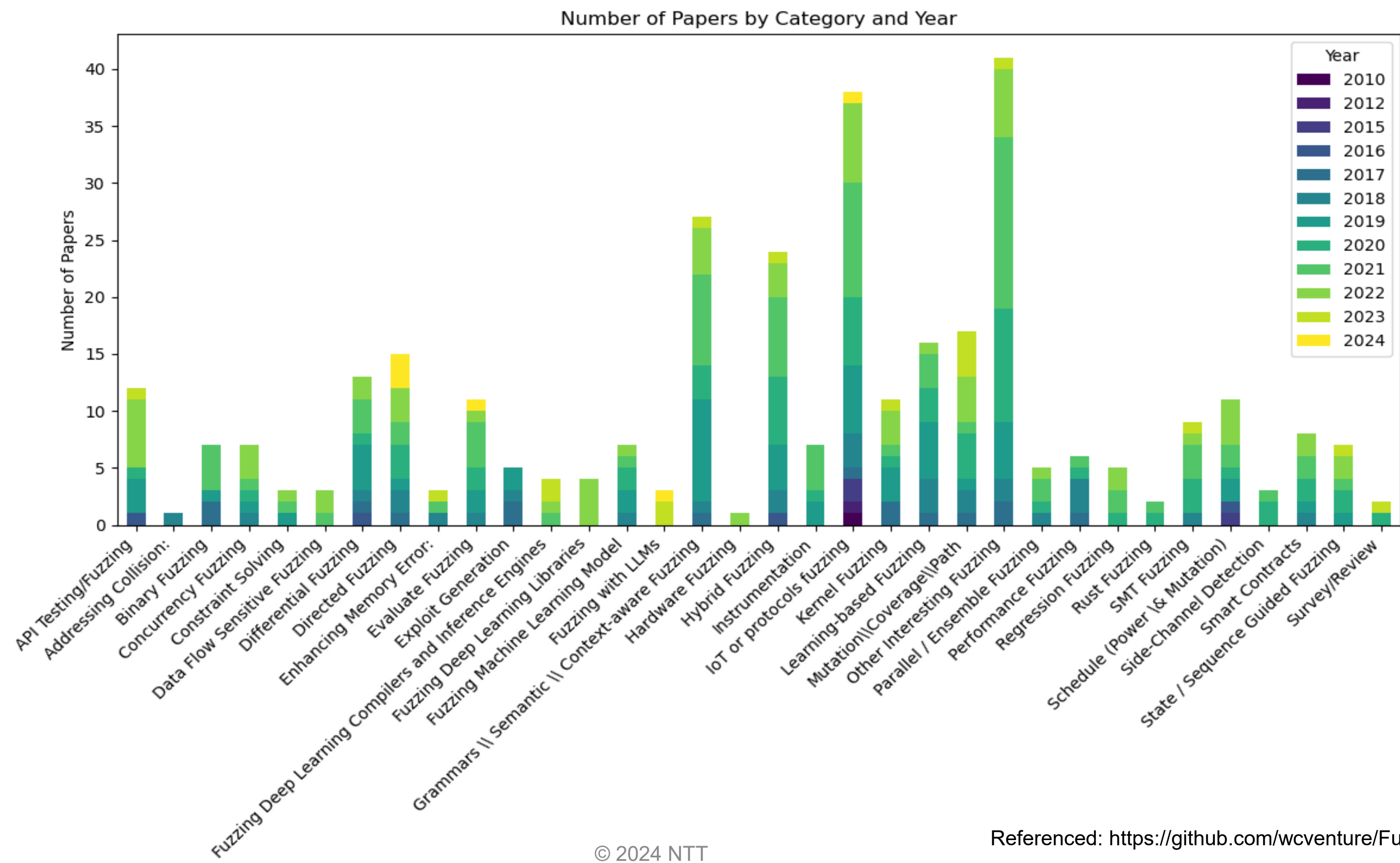
Type	Explanation	Tools
Black-box	It does not consider the internal state of TUT; it's synonymous with random testing.	Peach、SPIKE、Sully
Grammar-based	It allows the user to provide grammar rules that the input should follow, and modifies the input values accordingly.	Peach、SPIKE、Sully
Gray-box	It coarsely captures the internal state of TUT and uses it as a hint to adjust input values.	AFL、AFL++、libFuzzer、Honggfuzz
White-box	It analyzes while precisely capturing the internal state of the target under test. It is synonymous with symbolic execution	SAGE、KLEE、S2E
Hybrid	A combination of gray-box and white-box.	Driller、Qsym



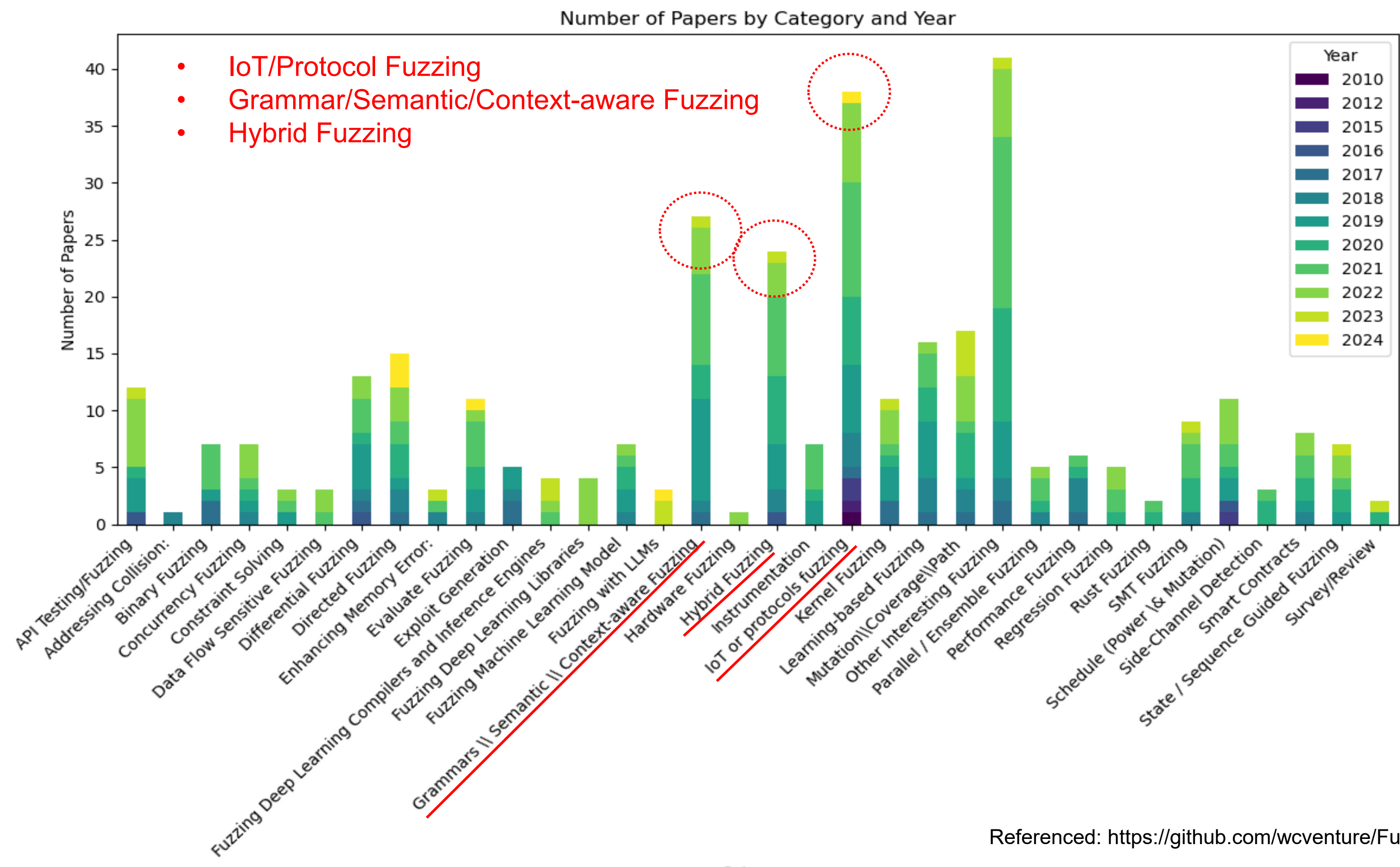
”Fuzzing: Hack, Art, and Science”, Patrice Godefroid, <https://cacm.acm.org/research/fuzzing/>

Complicated

# Academic Research on Fuzzing

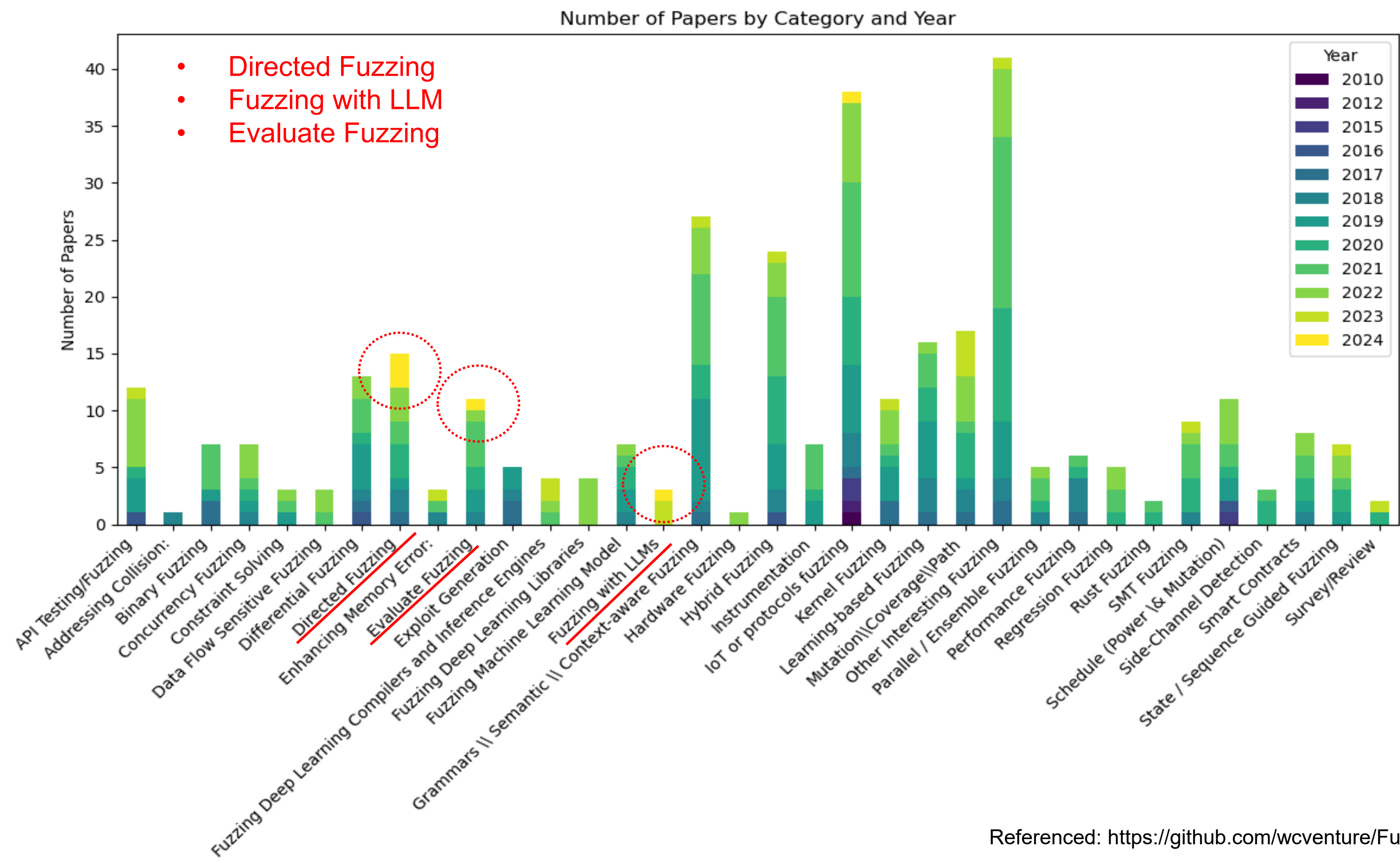


# Academic Research on Fuzzing: Hot Topic



Referenced: <https://github.com/wcventure/FuzzingPaper>

# Academic Research on Fuzzing: Recent Trend



Referenced: <https://github.com/wcventure/FuzzingPaper>

