

PkgFuzz Project: オープンソースソフトウェアのため の新たな継続的ファジング

川古谷 裕平 / 大月 勇人 / 塩治 榮太郎

2024
CODEBLUE
BECAUSE SECURITY MATTERS



自己紹介

川古谷 裕平

- NTTセキュリティホールディングス
- 特別研究員、博士（工学）
- マルウェア解析から脆弱性研究へシフトチェンジ中



大月 勇人

- NTTセキュリティホールディングス
- 主任研究員、博士（工学）
- 専門はメモリ解析、リバースエンジニアリング、OSセキュリティ



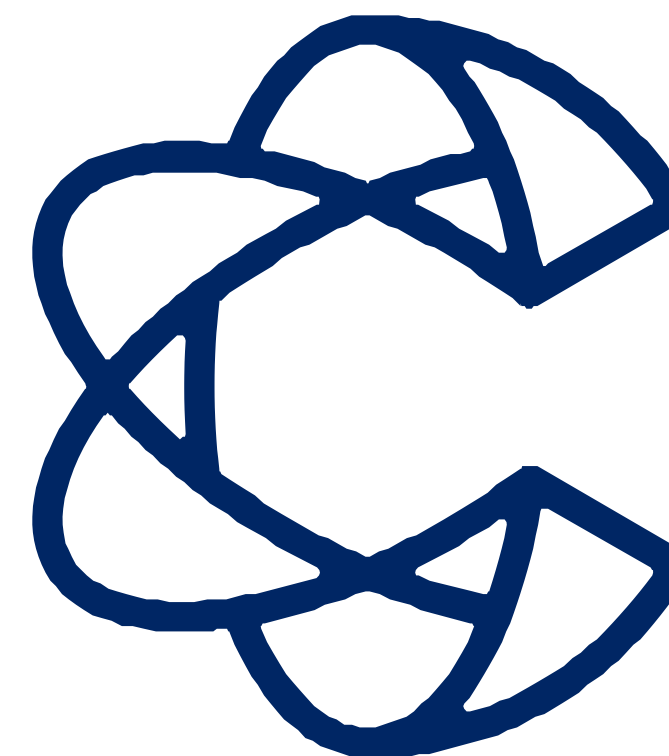
塩治 榮太朗

- NTT社会情報研究所
- 主任研究員
- 脆弱性発見・対策、ウェブ/モバイル/IoTセキュリティなどの研究や実用化開発に従事



発表の内容

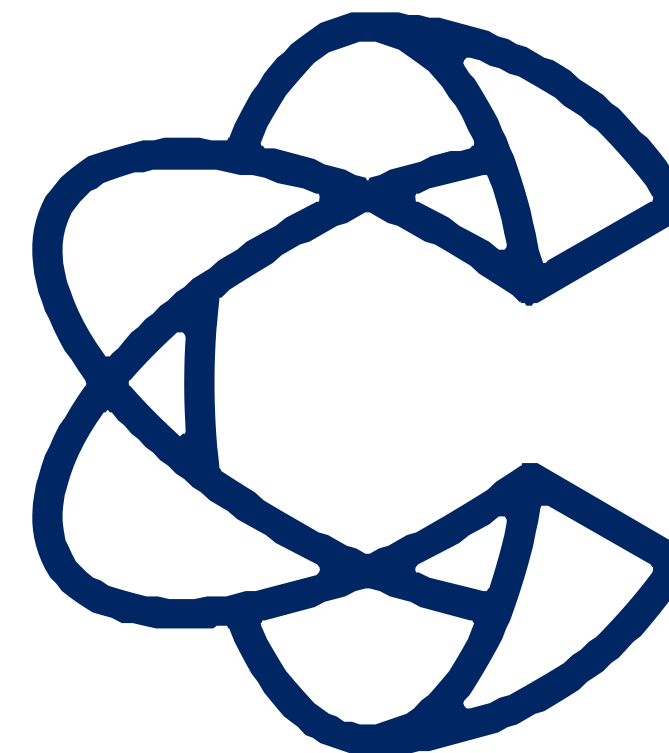
- 脆弱性とファジング
- PkgFuzz
- PkgFuzz Project
- クラッシュから脆弱性へ
- 今後の課題
- まとめ



発表の内容

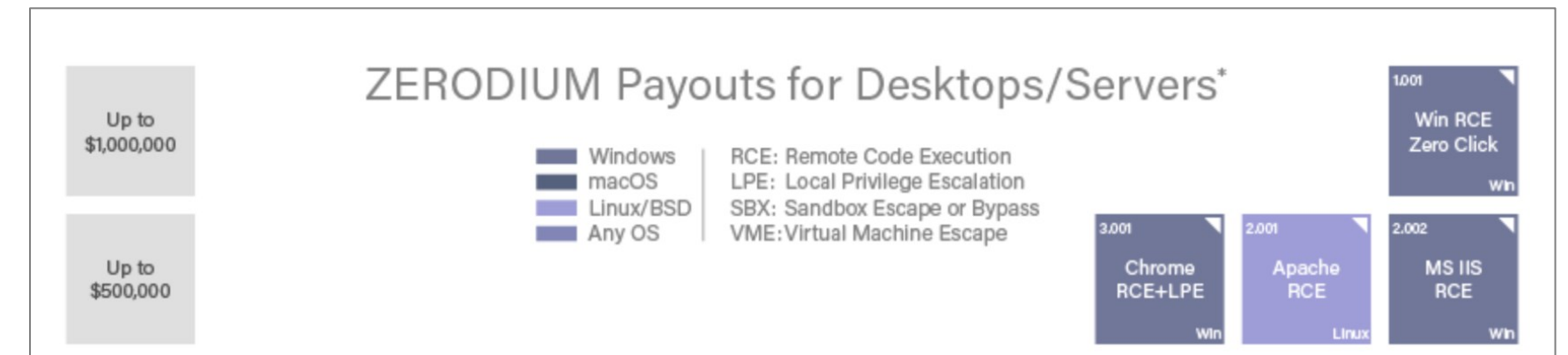
●脆弱性とファジング

- PkgFuzz
- PkgFuzz Project
- クラッシュから脆弱性へ
- 今後の課題
- まとめ



脆弱性の価値

- 脆弱性情報、特にユーザ数が多いプラットフォームのゼロデイ脆弱性の価値は高い
 - バグバウンティ
 - Exploit Acquisition Program
- ゼロデイ脆弱性にフォーカスしたハッキングコンテスト、Pwn2Ownの賞金総額は年々増え続けている
 - 2014 – 2017: \$460,000 to \$ 850,000
 - 2018 – 2023: \$1,000,000 to \$2,000,000



ZerodiumのExploit Acquisition Programでは、WindowsのRCE（リモート実行可能なExploit）に\$1Mの値が付く可能性がある*1

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

Pwn2Own Vancouver 2024で、優勝者は\$202,500 (≒2800万円)の賞金を獲得*2

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

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

脆弱性の見つけ方

過去 7 年分のBlack Hat USA Briefings概要AI分析

手法	説明	発表例
ファジング	ソフトウェアに無効または予期しない入力を送信して、クラッシュや異常な動作を引き起こすことによって脆弱性を発見する自動化された手法です。ソースは、ファジングがブラウザ、モバイルアプリケーション、ネットワークプロトコル、埋め込みデバイスなどのさまざまなターゲットで使用されていることを示しています。	2018 The Finest Penetration Testing Framework for Software-Defined Networks
リバースエンジニアリング	デバイスやソフトウェアの内部動作を理解するために、そのコードを分析します。ソースは、リバースエンジニアリングがファームウェア分析、カスタムプロトコルの解読、ハードウェアセキュリティメカニズムの理解などに使用されることを示しています。	2018 Exploitation of a Modern Smartphone Baseband
既知の脆弱性の分析	既知の脆弱性、エクスプロイト、攻撃手法を調査して、類似の脆弱性を発見します。ソースは、脆弱性の分析が、新しい脆弱性や攻撃ベクトルを発見するための貴重な情報源となり得ることを示しています。	2018 Return of Bleichenbacher's Oracle Threat (ROBOT)
サイドチャネル攻撃	デバイスのパフォーマンスや電力消費などのサイドチャネル情報を悪用して、機密データを取得します。ソースは、サイドチャネル攻撃が、特に組み込みデバイスやハードウェアセキュリティモジュールにおいて、深刻な脅威となる可能性があることを示しています。	2019 Dragonblood: Attacking the Dragonfly Handshake of WPA3
手動コード監査	この手法では、脆弱性を特定するために、セキュリティアナリストがソースコードを手動でレビューします。ソースは、手動コード監査が、ファジングなどの自動化された手法では発見が難しい、より複雑な脆弱性を発見するために使用されることが多いことを示しています。	2019 Lessons From Two Years of Crypto Audits

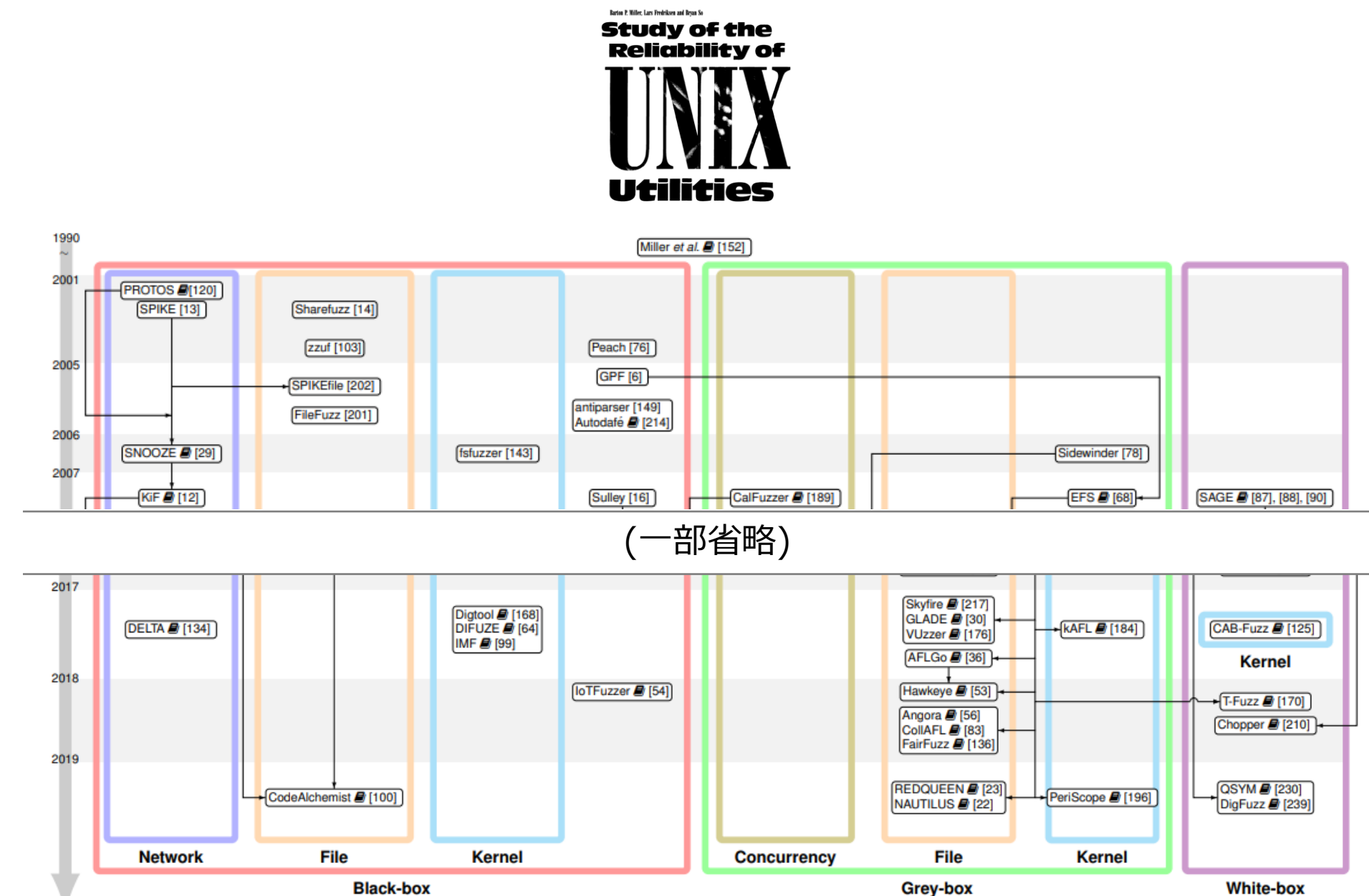
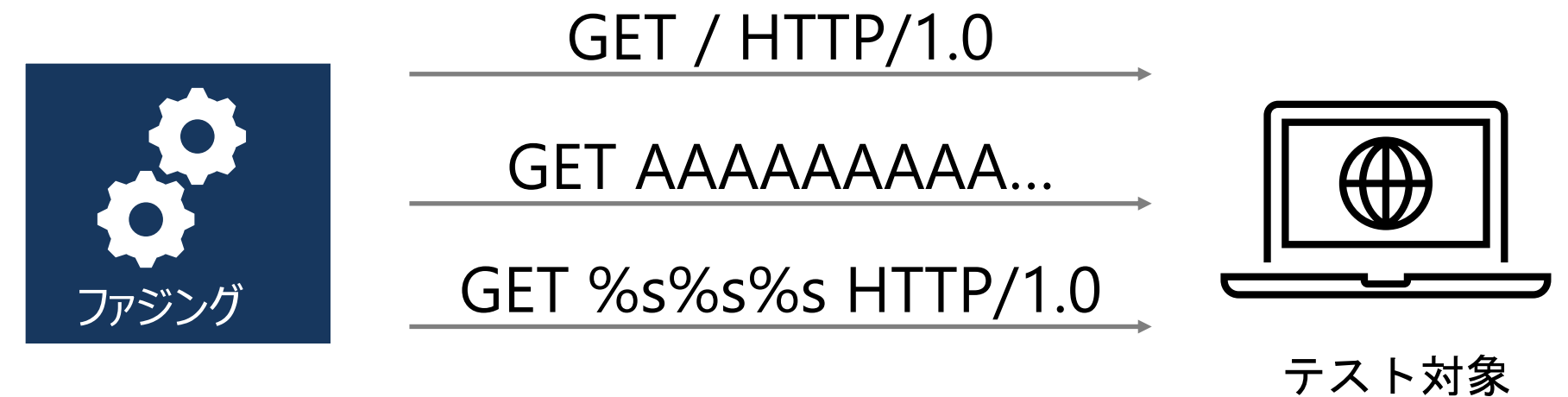
ファジング入門

■ ファジングとは？

- テスト対象のプログラムにランダムな値を与えて繰り返し実行し、そのプログラムをクラッシュさせる入力を自動的に見つけ出すテスト手法

■ 歴史

- “An empirical study of the reliability of UNIX utilities”, Communications of the ACM, 1990
- Barton P. Miller, 1988
 - 嵐による通信障害により壊れた値が各種UNIXコマンドに入力され、大量のクラッシュを引き起こしたことをヒントに考案した
- Millerの論文以来、多くの研究やファザーが発表されている



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

ファジングプロジェクト : Google OSS-Fuzz

■ 概要

- OSSを対象にしたファジングプロジェクト
- 最新のファジング技術とスケーラブルなクラウド環境を提供
- コミュニティの力を活用し、多様なソフトウェアへのファジングを実現

■ 参加条件

- “広範なユーザーベースを持つ、またはグローバルITインフラストラクチャにとって重要なOSSプロジェクトであること”
- 現在、1200+件程度

■ 成果（概要、2023年時点*1）

- 1000件の脆弱性修正、36000件のバグ

■ 課題

- コミュニティへの依存が大きい
- ...

(*1) <https://github.com/google/oss-fuzz>

(*2) <https://github.com/google/clusterfuzz>

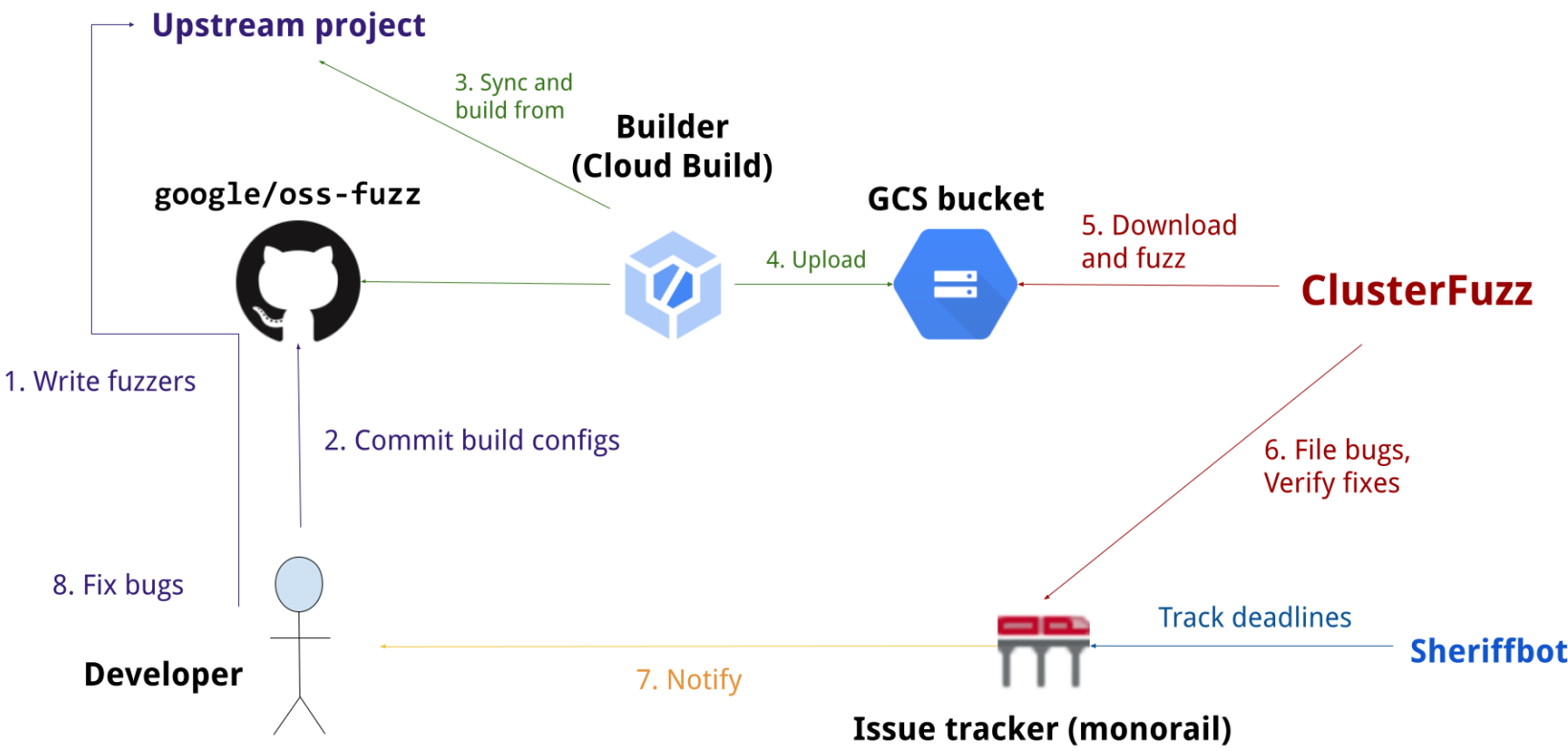
(*3) <https://issues.oss-fuzz.com/issues?q=status:open>

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_mb_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 xmlIFAParsePosCharGroup 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 librevenge::RVNGStringStreamPrivate::RVNGStringStreamPrivate ClusterFuzz Reproducible

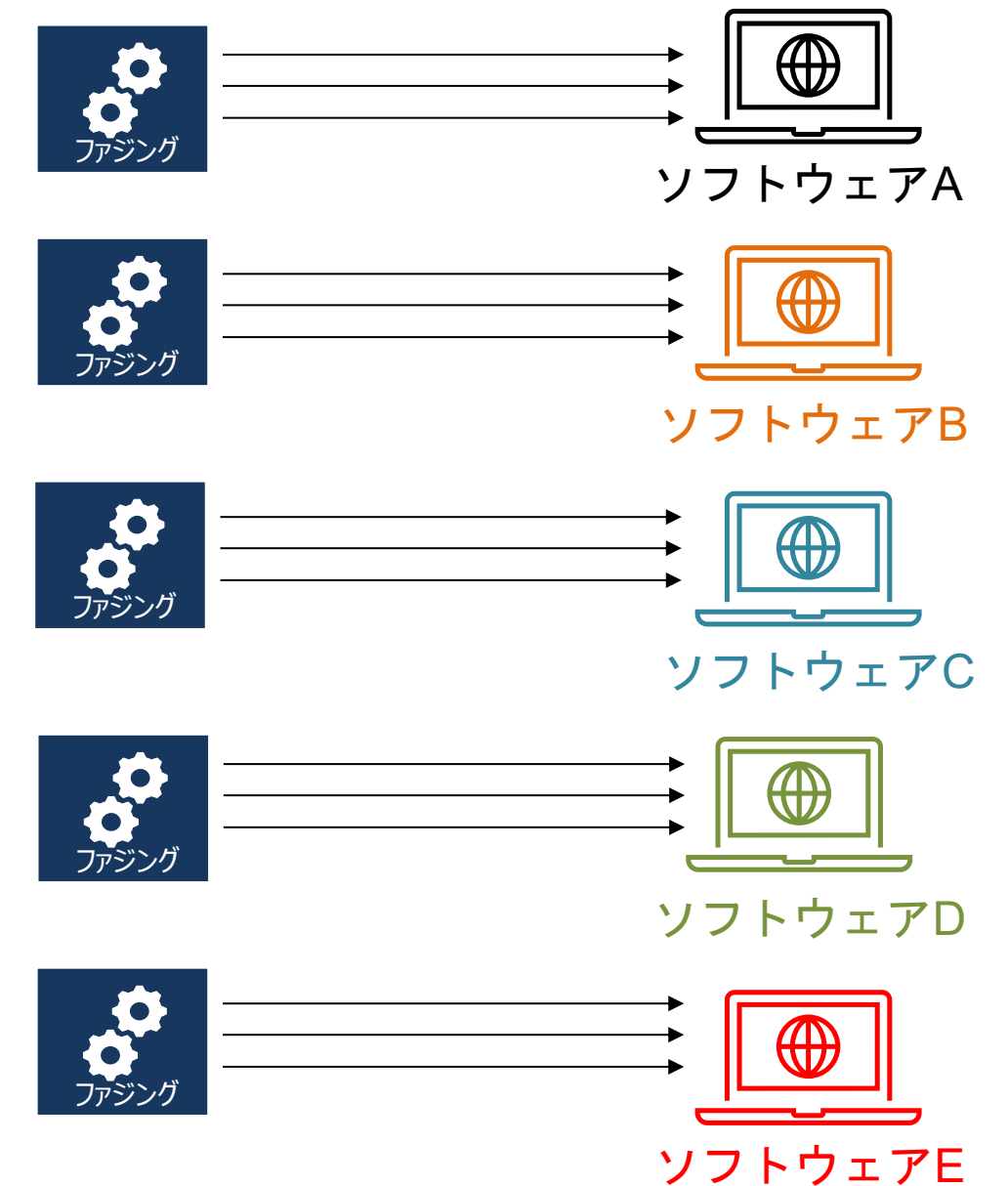
Issue Tracker (monorail)*3



ClusterFuzz: OSS-Fuzzの実行基盤*2

ファジングの自動化阻害要因

- ファジングは必ずしもワークフロー全体が自動化されているわけではない
 - 人間による前準備が重要
- ファジングを（効率的に）実施するために人間の介入が必要な個所
 - **#1 ハーネスの準備**
 - **#2 コマンドライン引数の調整**
 - **#3 ファジング用バイナリのビルド**
 - **#4 初期シードファイル**
- OSS-Fuzzでの対応
 - OSSプロジェクト側が個別に用意する
- これらを緩和できれば？
 - セキュリティテスト（ファジングテスト）に十分なリソースを割り当てられないOSSプロジェクト、主に小規模なOSSへファジングの恩恵、効果を届けることができる



異なるソフトウェアに対してファジングを継続的に実施する場合、人間による介入により効率が低下する。

自動化阻害要因：#1 ハーネスの準備

■ ファジング対象を呼び出すためのドライバ

■ AFLでそのままファジング可能な場合

- ファジング対象：main
- \$ afl-fuzz/readelf @@

■ ハーネスが必要な場合

- ファジング対象：ライブラリ関数
 - 例: zzip_file_open
- プログラムとして直接実行ができないので、ライブラリ関数を呼び出すハーネスを準備する
 - 例: zziptest.c
- \$ 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);
```

ハーネスの作成が必要ないケース*2

zziplib / bins / zziptest.c

Code

Blame

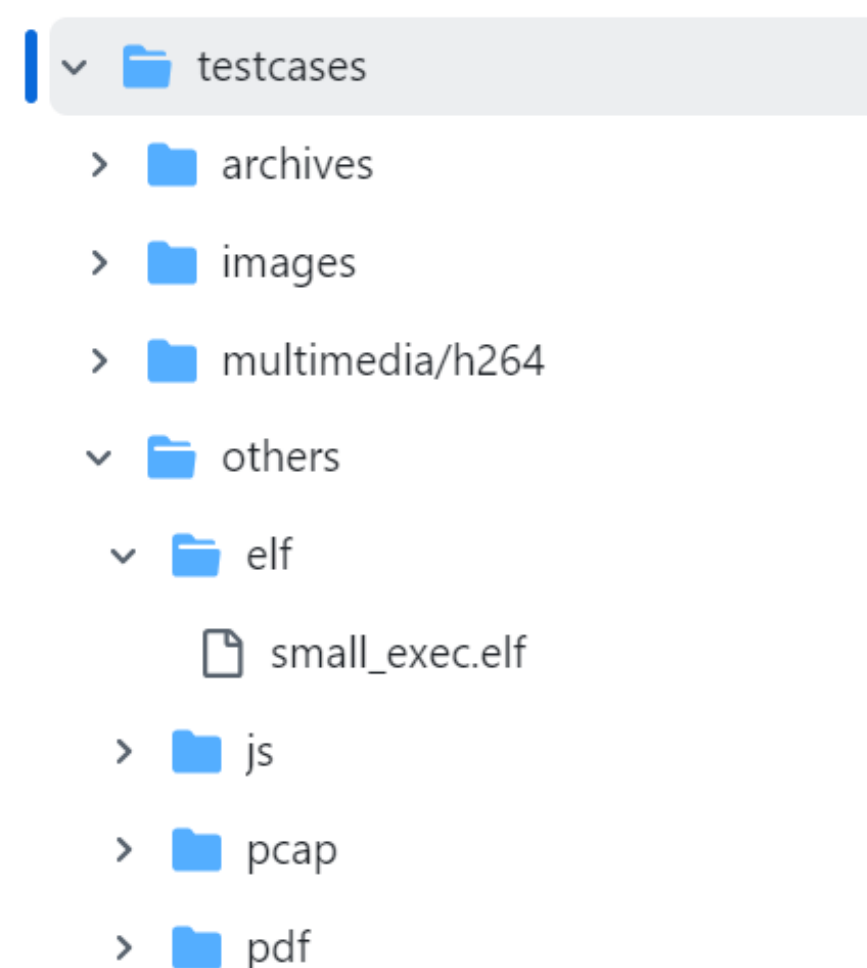
186 lines (165 loc) · 5.05 KB

```
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) { ライブラリ呼び出し部分
168             printf("error %d: %s\n", zzip_error(dir), zzip_strerror_of(dir));
169         }
```

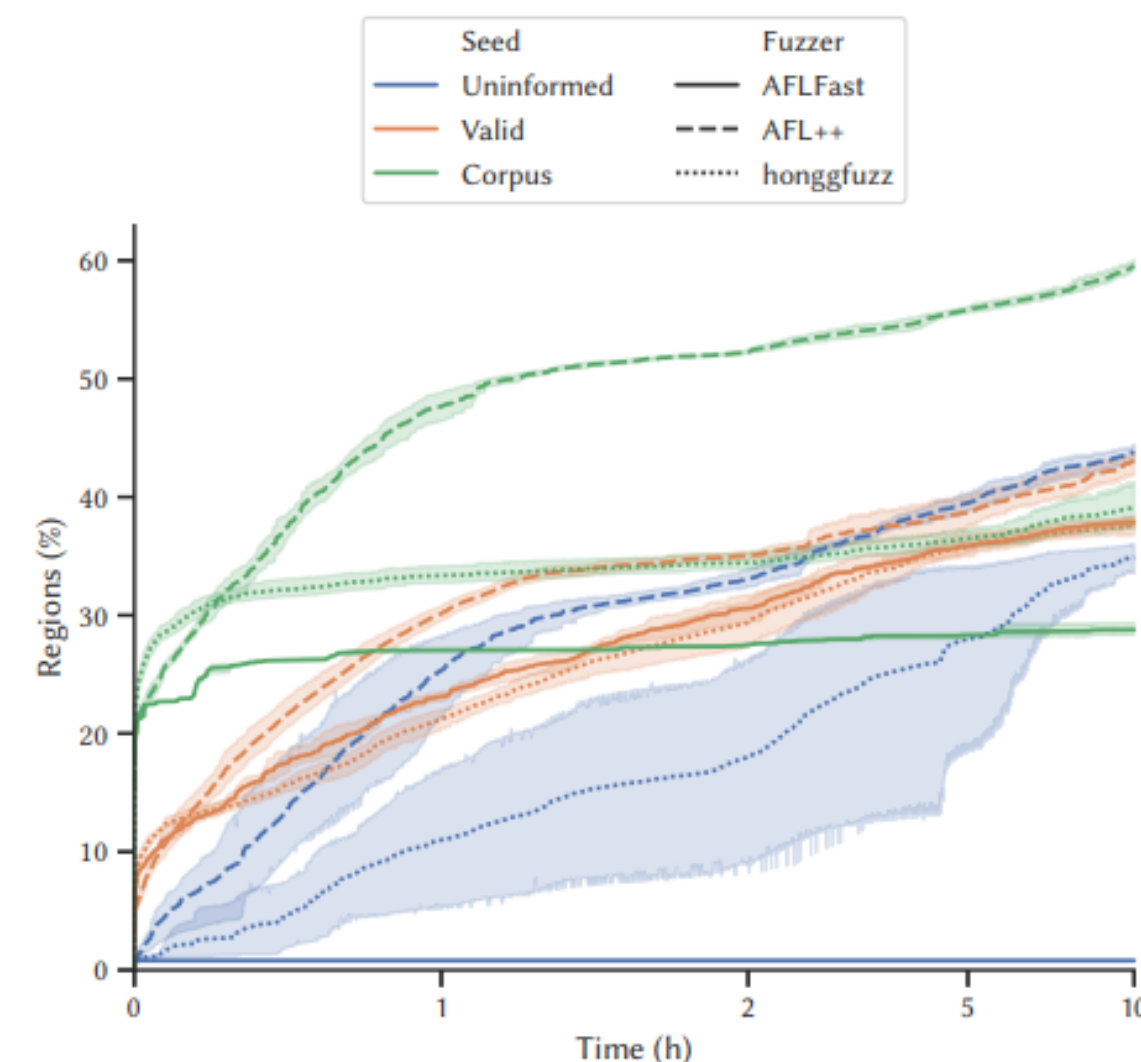
libzipのライブラリ関数(zzip_file_open)を呼び出すハーネス*1

自動化障害要因：#4 初期シードファイル

- 初期シードの有無によってファジングの効率（Coverage）は大きく変化する
- 従来はREADMEやmanpageを読むことで期待する入力を把握する
- インターネットを検索して、適切なシードを準備する
 - AFL++ Testcases^{*2}
 - fuzzing-seeds^{*3}



AFL++ Testcases: 各種フォーマットのサンプルファイルが準備されている



readelf に対するファジングのカバレッジ比較結果^{*1} : 適切なシード(Corpus:シードファイルのセット)を用意した場合、AFL++で約1.5倍の結果の違いが出た

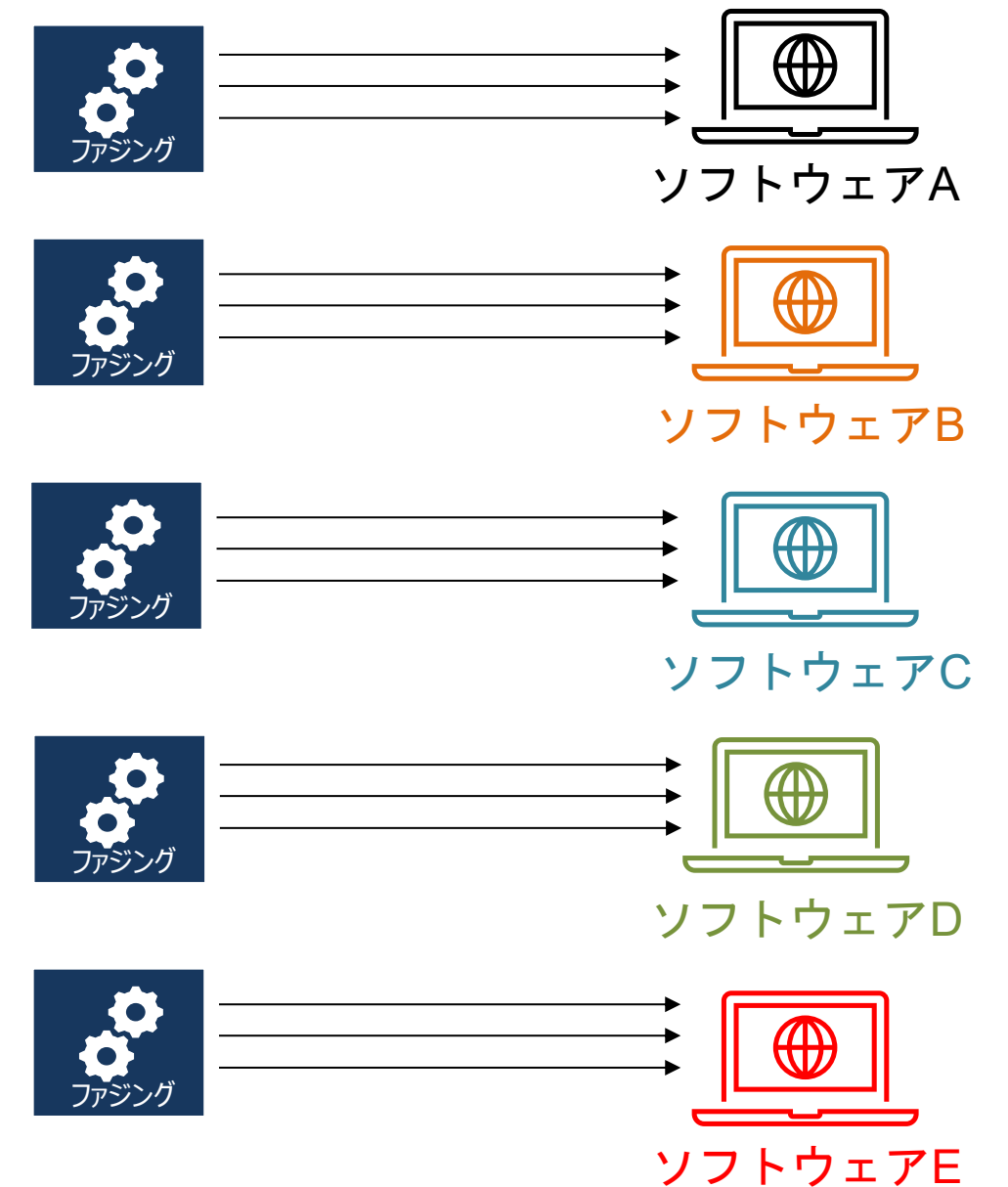
(*1) "Seed selection for successful fuzzing", Adrian Herrera et al. <https://dl.acm.org/doi/10.1145/3460319.3464795>

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

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

【再掲】ファジングの自動化阻害要因

- ファジングは必ずしもワークフロー全体が自動化されているわけではない
 - 人間による前準備が重要
- ファジングを（効率的に）実施するために人間の介入が必要な個所
 - #1 ハーネスの準備
 - #2 コマンドライン引数の調整
 - #3 ファジング用バイナリのビルド
 - #4 初期シードファイル
- OSS-Fuzzでの対応
 - OSSプロジェクト側が個別に用意する
- これらを緩和できれば？
 - セキュリティテスト（ファジングテスト）に十分なリソースを割り当てられないOSSプロジェクト、主に小規模なOSSへファジングの恩恵、効果を届けることができる



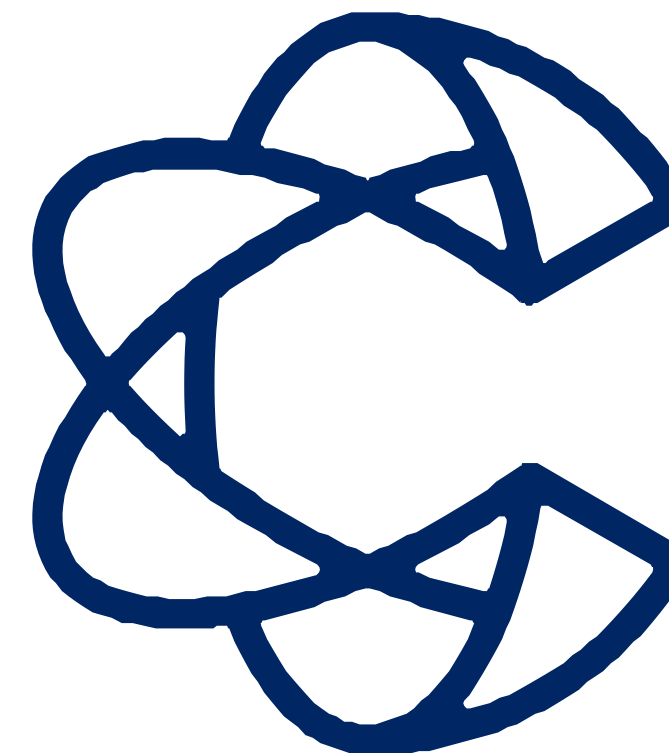
異なるソフトウェアに対してファジングを継続的に実施する場合、人間による介入により効率が低下する。

発表の内容

- 脆弱性とファジング

● PkgFuzz

- PkgFuzz Project
- クラッシュから脆弱性へ
- 今後の課題
- まとめ



基本アイデア

- ソフトウェアパッケージ(.deb等)のビルドプロセスを監視し、ファジングに必要な情報を自動収集する

■ 利点

- パッケージのテストとして、ライブラリ関数を呼び出すテストコードが含まれる
 - 「#1 ハーネスの準備」
- パッケージのテストを監視することで、適切なコマンドラインオプションや入力ファイルがわかる
 - 「#2 コマンドライン引数の調整」、
「#4 初期シードファイル」
- ビルドオプションの渡し方等が共通化されており、オプションの追加が容易
 - 「#3 ファジング用バイナリのビルド」

```
$ 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)
```

パッケージのソースのダウンロード

ソフトウェアのビルド

ビルドの過程でテストも実行される

環境変数等でビルド方法をカスタマイズ可能

PkgFuzz

- 基本アイデアを実装し、ファジングワークフロー全体を自動化したシステム
 - 3つのフェーズから構成：パッケージ解析、ファジング、クラッシュトリアージ^{*1}
 - パッケージの URL を入力するだけで、トリアージ済みクラッシュ入力が蓄積される



(^{*1}) **クラッシュトリアージ**：大量のクラッシュを分類し、後続の解析で優先すべき対象を選出する。

フェーズ1：パッケージ解析

- 1 ソフトウェアパッケージ（ソースパッケージ）を取得し、ビルドする
- 2 ビルド過程を監視し、以下の構成情報を収集
 - 実行ファイル
 - コマンドライン引数
 - テスト用の入力ファイル
- 3 パッケージのビルドオプションにて、ファジング用コンパイルオプションを指定し、ファジング用バイナリを生成する

使用ツール: execsnoop改良版 (BPF/eBPFを利用してexec系システムコールをフックするツール)



例: jbigkit v2.1-6

1 execsnoopでビルドプロセスを観測し、ELF実行が確認できたログを抽出

実行時刻	コマンド名	PID	PPID	execveの戻り値	コマンド引数
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.jpg" "test-ccitt1.pbm"
02:10:46	jbgtopbm	3443723	3443666	0	"/jbgtopbm" "test-t82-koXoyx5U.jpg85" "test-t82-koXoyx5U.pbm85"
02:10:46	pbmtojpg	3443686	3443655	0	"/pbmtojpg" "test-ccitt1.pbm" "test-ccitt1.jpg"
02:10:46	pbmtojpg85	3443672	3443666	0	"/pbmtojpg85" "-p" "0" "test-t82.pbm" "test-t82-koXoyx5U.jpg85"

2 ファイルを引数に取っているものを抽出

実行時刻	コマンド名	PID	PPID	execveの戻り値	コマンド引数
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.jpg" "test-ccitt1.pbm"
02:10:46	jbgtopbm	3443723	3443666	0	"/jbgtopbm" "test-t82-koXoyx5U.jpg85" "test-t82-koXoyx5U.pbm85"
02:10:46	pbmtojpg	3443686	3443655	0	"/pbmtojpg" "test-ccitt1.pbm" "test-ccitt1.jpg"
02:10:46	pbmtojpg85	3443672	3443666	0	"/pbmtojpg85" "-p" "0" "test-t82.pbm" "test-t82-koXoyx5U.jpg85"

3 最終的にファジングに投入したコマンドライン

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

コマンドのパスを補完

コマンドライン中の入力ファイルを特定
(Fuzzerに位置を通知するため"@"に置換)

フェーズ2：ファジング

- 1 短時間のファジングを実施
 - エラーの有無、カバレッジの伸び率(=コーパスが拡大状況)を確認
- 2 上記を確認後、長期間ファジングを実施

使用ツール: AFL++ 4.08c (ファザー) + Address Sanitizer (サニタイザー)



フェーズ3：クラッシュトリアージ

- 1 同じバグに関連するクラッシュを分類・除去する
- 2 攻撃可能性を簡易的に判断し、優先度付けを行う
- 3 オリジナルのバイナリで再現性を確認する

使用ツール: CASR (Crash Analysis and Severity Report) + AFLTriage



CASR (Crash Analysis and Severity Report)

- クラッシュレポートを収集し、トリアージ、深刻度の評価を行うツール
 - 各種評価ロジックはgdbのexploitableプラグインのものがベース
- 種類: シグナルの情報やサニタイザの出力から判定
 - AbortSignal, stack-buffer-overflow(write)など
- 深刻度: 種類に基づいて3段階に分類される
 - 高 (**EXPLOITABLE**): SegFaultOnPc, Write BoF など
 - 制御を容易に奪えると考えられるクラッシュ
 - 中 (**PROBABLY_EXPLOITABLE**): BadInstruction, 上記以外の BoF など
 - 多少の追加解析で制御を奪えるか判定が可能と思われるクラッシュ
 - 低 (**NOT_EXPLOITABLE**): AbortSignal, double-free, Read BoF など
 - 制御を奪えるとは言い難いクラッシュ

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

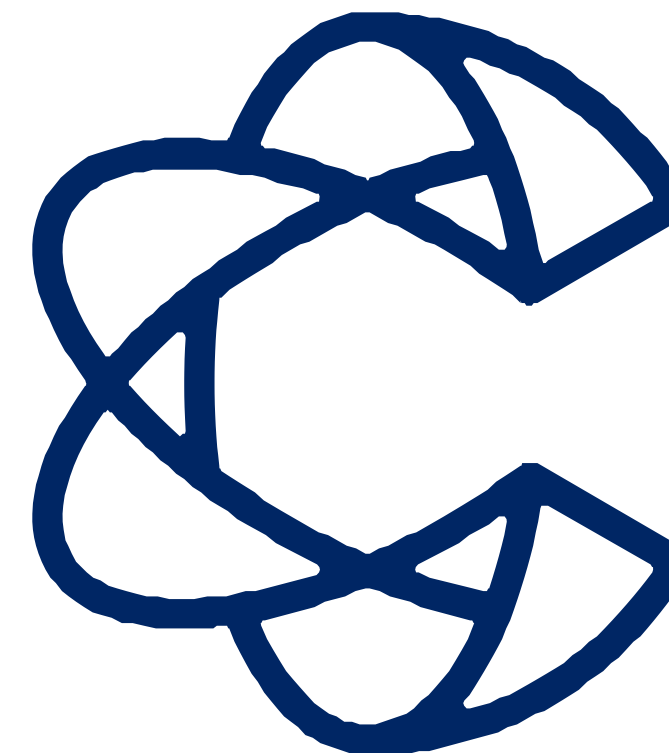
発表の内容

- 脆弱性とファジング

- PkgFuzz

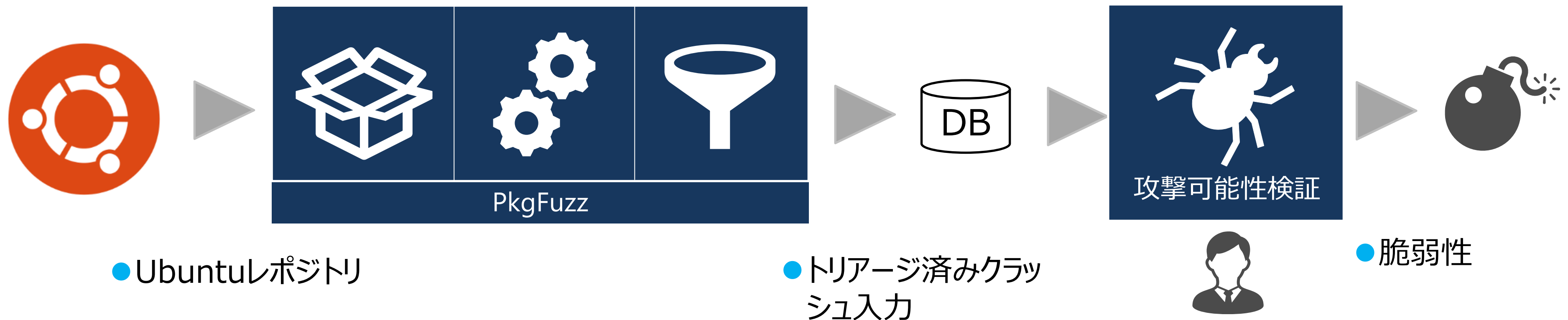
● PkgFuzz Project

- クラッシュから脆弱性へ
- 今後の課題
- まとめ



PkgFuzz Project

- **目的** : Ubuntuパッケージの脆弱性調査とPkgFuzzの性能評価
- **対象** : Ubuntu 23.10の265件のDebianパッケージ
- **手段** : PkgFuzzを利用したファジングと人間による攻撃可能性検証
 - PkgFuzzにより自動的にクラッシュ入力を作成
 - PkgFuzzの出力を人間が精査し、Exploitコードを作成して検証する



パッケージの選出

フィルタ条件		パッケージ数
1	Ubuntuパッケージ総数	71,858
2	データ取得ができたパッケージ件数	71,277
3	除外リストのパッケージを除いた件数 ^{*1}	46,585
4	ソースパッケージ単位に集約した件数	24,135
5	C/C++のLOCが1Ks以上のもの ^{*2}	10,116
6	ChangeLogの最初が2015/01/01以前 ^{*3}	6,915
7	PkgFuzzの選出過程で長期間ファジングまで残ったもの ^{*4}	265

(^{*1}):Linuxカーネルや言語処理系など、fuzzing対象として扱うのが難しいものは除外する
(^{*2}):コード量の少ないプロジェクトはバグが混入する可能性が低いと考え、1K以上を対象とする
(^{*3}):新しいプロジェクトは最新のソフトウェアエンジニアリングやツールが利用されるため、バグが混入する可能性は低いと考え、古いコードが含まれるものを対象とする
(^{*4}):パッケージ解析フェーズでのビルドに失敗するものや、ファジングフェーズの短時間ファジングでカバレッジが伸びないものは除外する

PkgFuzz Projectの結果概要

■ 318日間(2023/9/19 ~ 2024/7/24)のPkgFuzzによるファジングにより以下の結果を得た

項目	結果
長期間ファジング対象パッケージ	265パッケージ
ファジングを実行した回数の合計	606回 (2.28回/1パッケージ ^{*1})
合計クラッシュ数	64,658件
トリアージ後のクラッシュ数(AFLTriage)	1,186件(54.5クラッシュ/1バグ ^{*2})

(*1):同じパッケージに対して異なるコマンドライン引数が観測された場合、複数回のファジングが試行される
(*2):AFL++はカバレッジを拡大した入力をシードに追加するため、クラッシュした入力に類似したものも生成されやすい、と考えられる

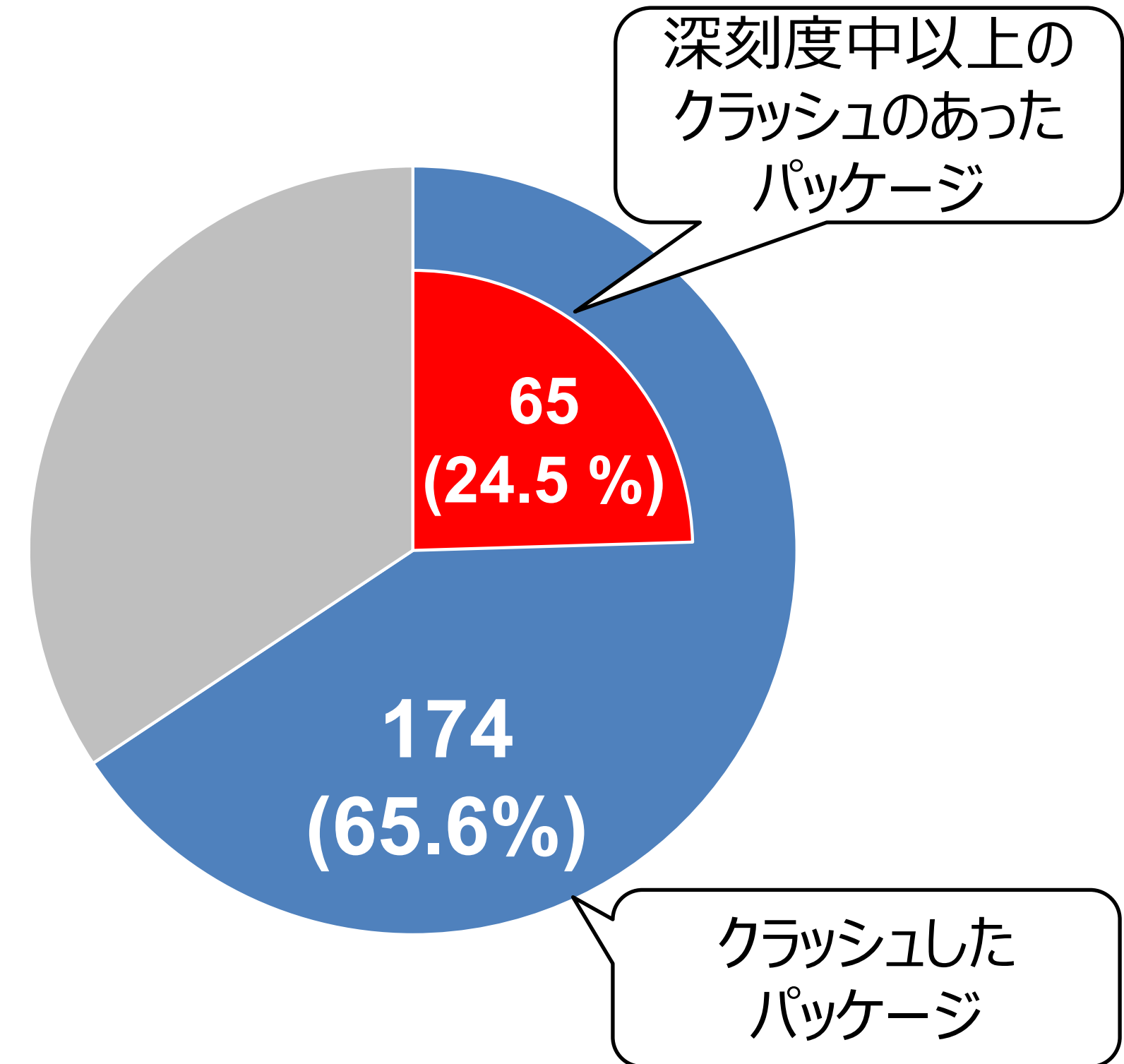
結果分析

■ PkgFuzz Projectでの結果を元に以下の観点で分析を行う

- 脆弱性を含むパッケージの割合
 - クラッシュが観測されたパッケージの割合
 - クラッシュの原因
- PkgFuzzによるファジングの効率性
 - 人間によるファジング（OSS-Fuzz）との比較
 - クラッシュが観測されないパッケージの特徴
 - 適切なファジング実行時間

脆弱性を含むパッケージの割合

- クラッシュが観測されたパッケージの割合
→ 1つ以上のクラッシュが得られたパッケージは
174パッケージ(=65.6%)
 - 約2/3のパッケージにはバグが存在する
- 脆弱性が存在する可能性が高いパッケージの割合
→ 深刻度中以上のクラッシュを1つ以上含むものは
65パッケージ(=24.5%)
 - 約1/4のパッケージには脆弱性が存在する可能性が高い

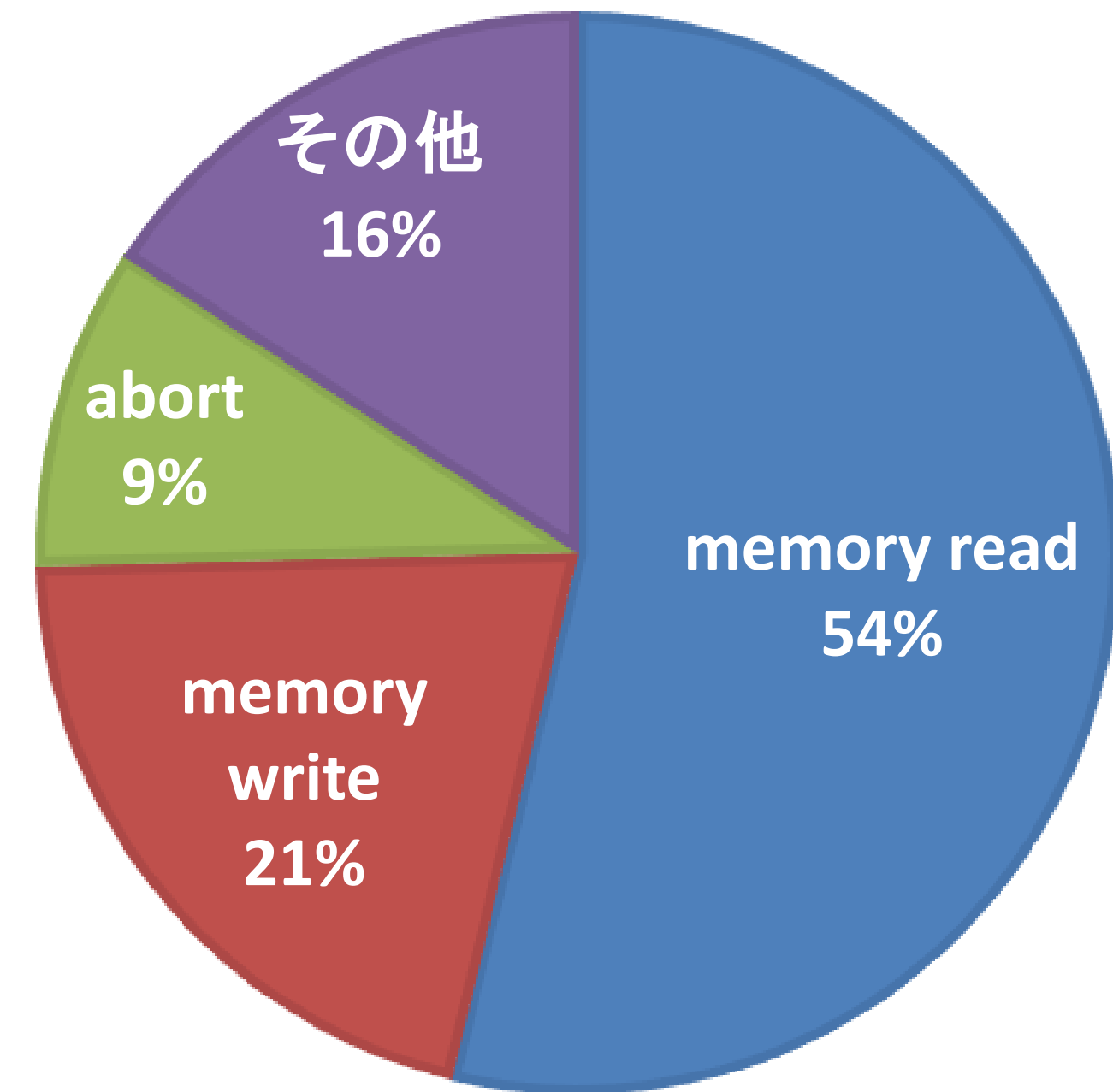


脆弱性を含むパッケージの割合

■クラッシュの原因

→CASRによれば、主な原因は**メモリ読み書き**

- メモリ読み込み: **1123件 (全体の54%)**
 - 読み込みはクラッシュの原因だが、読み込みしかできないか否かは本来はさらなる調査が必要^{*1}
- メモリ書き込み: **435件 (全体の21%)**
 - 脆弱性に直結しやすいことから、深刻度は高く評価される



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

PkgFuzzによるファジングの効率性

- 人間によるファジング（OSS-Fuzz）との比較による評価
 - PkgFuzz ProjectとOSS-Fuzzで共通しているOSS(パッケージ)数: 32 (約12%)
 - ファジングによって実行されたコードのカバレッジとバグ検出数の観点で比較
- 結果
 - カバレッジ
 - 行カバレッジを計測できた共通のOSSの数：18
 - 平均では14ポイントの差があるが、PkgFuzzの方がカバレッジが伸びたケースもある
 - バグ検出数
 - OSS-Fuzzの方が1 OSSあたりのバグ検出数は多いが、脆弱性率に大きな差はない

	平均 行カバレッジ	バグ検出数		脆弱性率
		合計	平均バグ数	
OSS-Fuzz	54.65%	1000>	21.2	26%
PkgFuzz	40.80%	1186	4.47	18%

パッケージ名	OSS-Fuzz	PkgFuzz
libpsl	19.22%	51.01%
speex	35.09%	43.56%
ninja	36.09%	39.26%
...

参考：OSS-Fuzzの情報はGoogleのブログやissue trackerなどから取得
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>

PkgFuzzによるファジングの効率性

- クラッシュが観測されないパッケージのパターン：
カバレッジの伸びが早期に飽和状態

- 例：w3m

- 原因分析：
ファジングブロック要因の存在（Fuzz Introspector）

- マジックナンバー、チェックサム、キーワードなど

- 改善案

- 特性の異なるファザーも組み合わせるアンサンブルファジング
（例：AFL++、libFuzzer、Honggfuzz）や、
プログラム解析技術を組み合わせるハイブリッドファジングにより、
ファジングの効率性やブロック要因を解消する

8.57%

ファジングをおよそ6日間実施して
bitmap_cvg (edge coverage)の値
がほぼ変動なし(終了時で8.57%)

クラッシュ数 0

```
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;
```

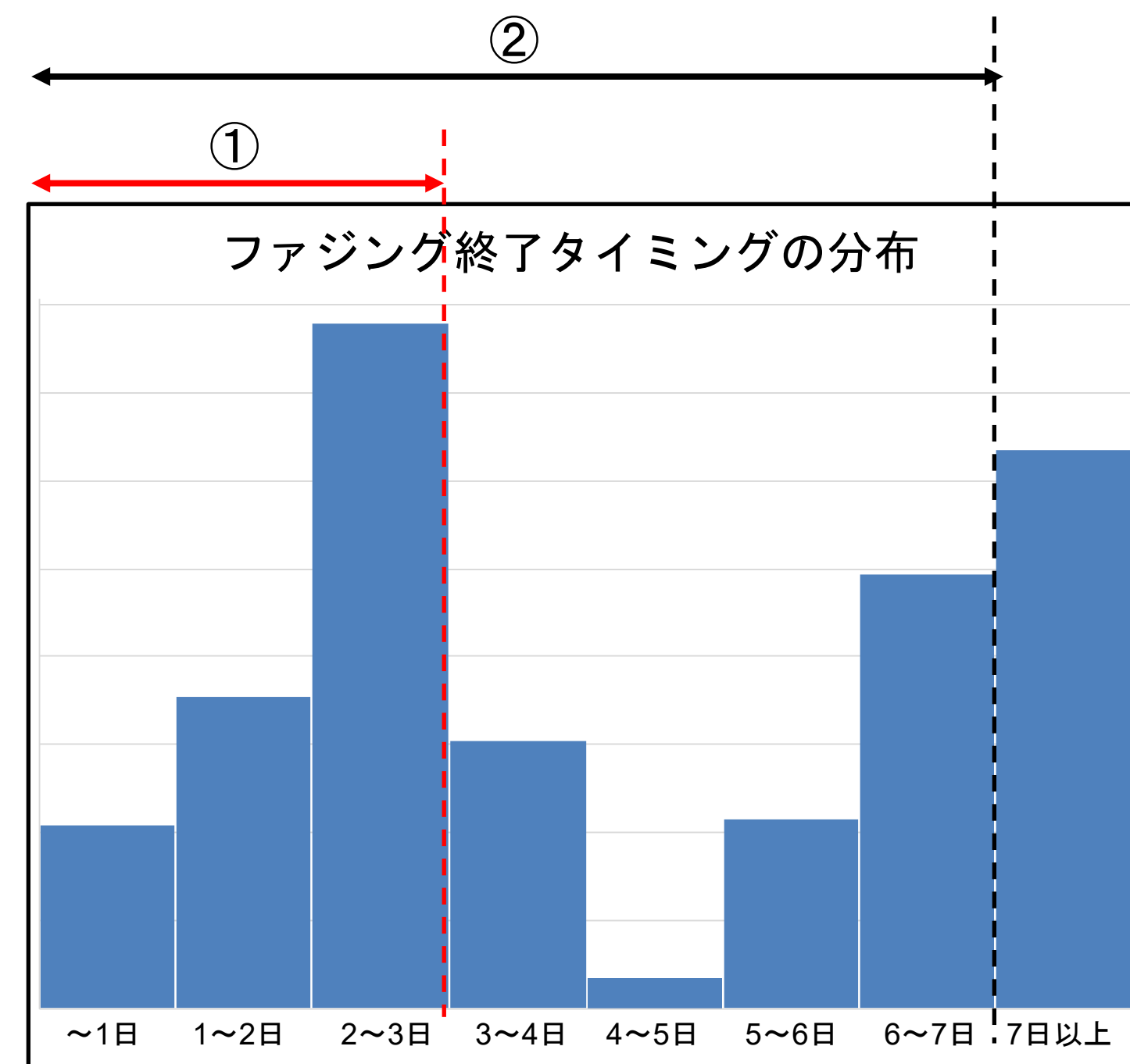
PkgFuzzによるファジングの効率性

■ 適切なファジング実行時間

- 3日以内に**全体の44%(①)**が終了し、**1週間で79%(②)**が終了
- PkgFuzz Projectでのファジング終了基準
 - 基本設定は1週間 (経験則)
 - + カバレッジ拡大状況、並列数等から人間が判断
 - ファジング時におけるカバレッジの上昇傾向^{*1}を考えると、今回の基準は最適ではない

■ 改善案

- 既存研究^{*2*3}を参考に、検査すべき箇所をあらかじめ特定した上で、そこに対する誘導やカバレッジ計測を行うことを検討予定



(^{*1})「時間に比例して徐々に上昇するのではなく、突然上昇し、その後はほぼ同じ状態が一定期間継続する」といった挙動を繰り返す、と言われている
(断続平衡 : 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

PkgFuzzによる結果分析のまとめ

■ 脆弱性を含むパッケージの割合

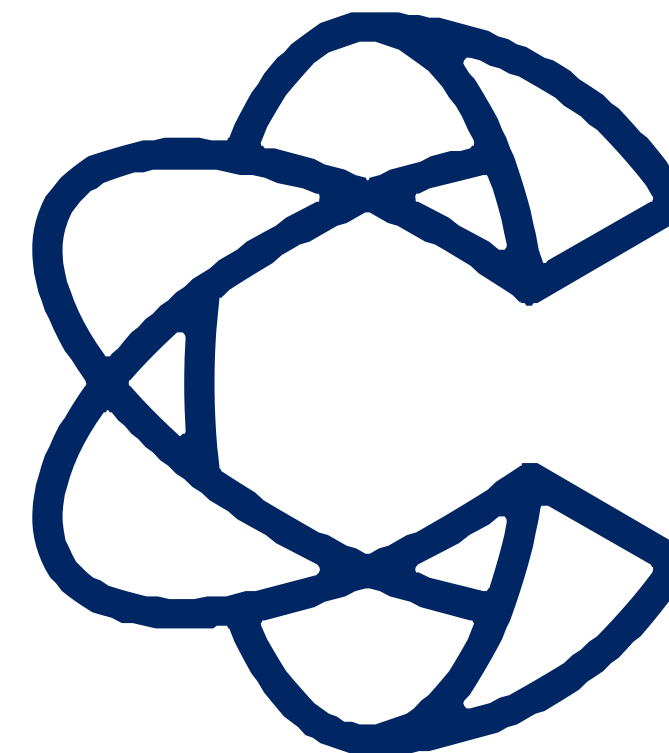
- 脆弱性を含む可能性のあるパッケージは1/4程度 (=65/265)
- しかし、メモリ読み込みに起因するクラッシュは脆弱性の深刻度を過少評価している可能性がある

■ PkgFuzzによるファジングの効率性

- OSS-Fuzzと比較したところ、PkgFuzz Projectでは、人間がファジングを行った場合と同等の結果を自動的（人間の介入なし）に実施できた
- 自動化したことにより、1OSSに対するファジングを低リソースで可能にした。これにより、OSS-Fuzzが対象としていないOSSに対してファジングを実行でき、多くのバグを検出した（OSS-Fuzzとの重複はわずか12%=32/265）
- ブロック箇所の回避や終了条件の設定に関して、改善の余地がある。既存研究の成果をPkgFuzz Projectへ取り込み、より効率なファジングシステムを設計する

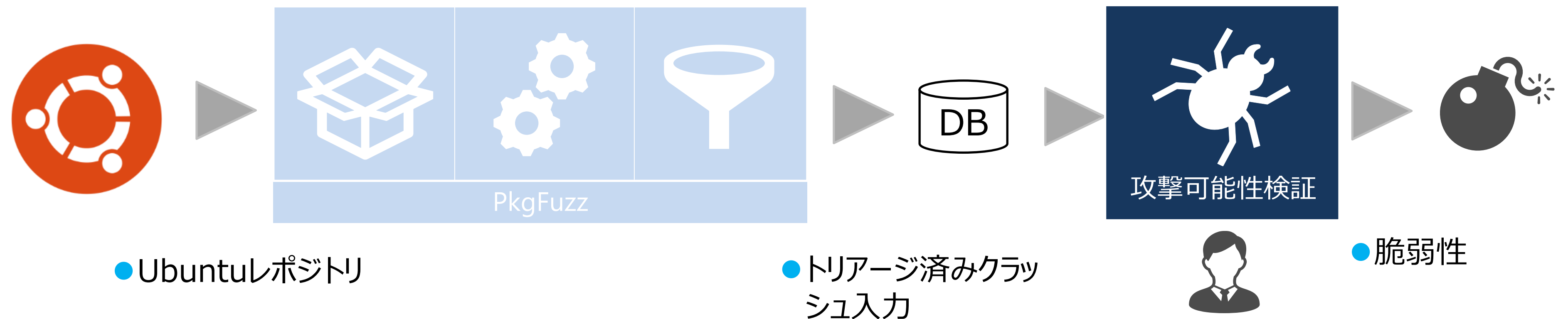
発表の内容

- 脆弱性とファジング
- PkgFuzz
- PkgFuzz Project
- クラッシュから脆弱性へ
- 今後の課題
- まとめ



攻撃可能性検証

- クラッシュがトリアージツールによって攻撃可能性あり（exploitable/probably exploitable）と自動的に判定されたとしても、あくまでも推測であるため、より詳細に検証する必要がある
- 攻撃可能性ありと判定された65件のパッケージ（484件のクラッシュ入力）に関して、実際に制御が奪えるかの手動での検証を行った
 - ステップ1: ユースケース調査（65件→22件）
 - ステップ2: Exploitability調査（22件→4件）



ステップ 1 : ユースケース調査

- **目的** : クラッシュが発生した状況におけるプログラムのユースケース（利用方法）を調査し、攻撃の実現可能性に基づいた絞り込みを行う
- **手法** : パッケージのドキュメントなどから手動でユースケースを推定し、攻撃者が外部から入力を制御できない状況で利用されている場合は、Exploitできたとしても実際に攻撃することは難しいため除外する
 - 除外例 :
 - ビルドプロセスの一環として使用されるもの
 - サーバソフトウェアの設定ファイルをパースするもの
- **結果** : パッケージ65件から非現実的なユースケースを除外し22件に絞りこんだ

ステップ 2 : Exploitability調査

- **目的** : クラッシュがExploitableであることを、実際に制御が奪うことができるExploitコードを作成することによって示す
 - トリアージツール（CASR）の機械的な判定は、あくまでも参考情報程度のため
- **手法** : ソースコードの詳細調査やデバッガを利用した調査によって、バグの根本原因を特定しつつ、Exploitコードの作成を手動で実施した
- **結果** : パッケージ22件のうち4件（クラッシュ5件）でExploitコードの作成に成功した
 - 脆弱性の内訳 : Stack Buffer Overflow 3件、Heap Buffer Overflow 2件
 - その他 : 脆弱性として利用することが難しい、ソフトウェアの仕様、など

例：Assimp(CVE-2024-40724)

■ Open Asset Import Library

- 様々な3Dファイルフォーマットを、プログラムで扱うことのできるフォーマットに変換するためのライブラリ

■ 攻撃シナリオ例

- 3DモデルデータをAssimpを使ってロードするプログラムを利用するユーザに対し、攻撃者は細工されたply形式ファイルを読み込ませる（インターネット配布、SE攻撃等）

■ 脆弱性の原因

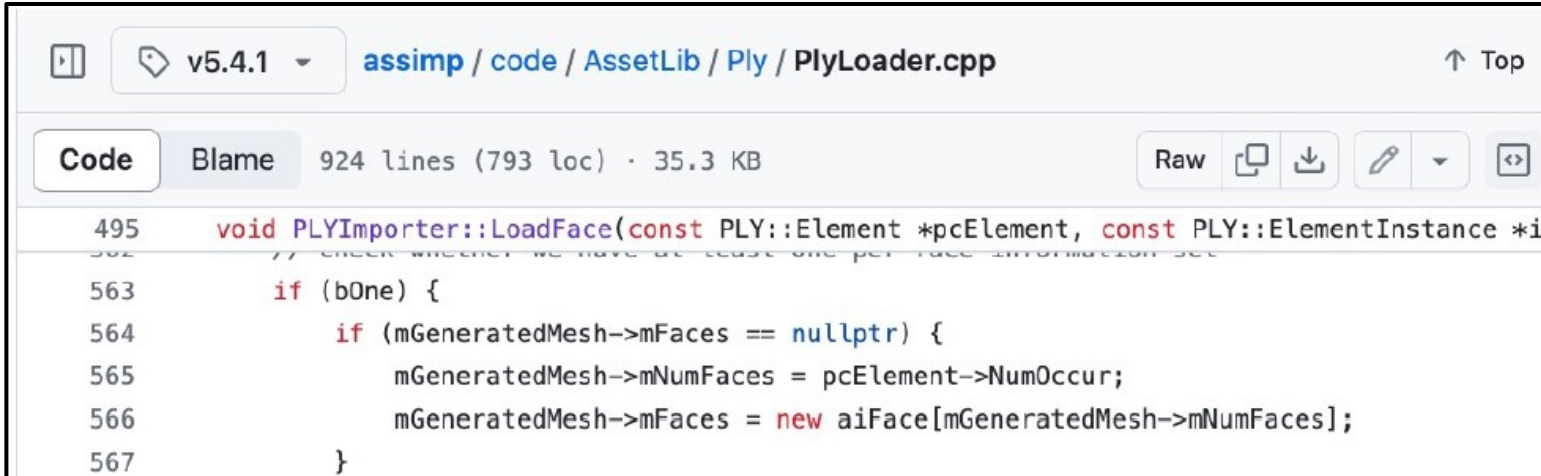
- Assimpのplyファイルのロード処理において、メモリ領域の確保方法に欠陥があり、要素を複数回読み込む際にHeap Overflowが可能になる

■ Exploit方法

- Heap Overflowにより隣接するメタデータを破壊することで、任意アドレス書き換えが可能となり、それを利用してポインタを上書きしShellを取得できる

```
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
...
```

plyファイルの例



```
assimp / code / AssetLib / Ply / PlyLoader.cpp
Code Blame 924 lines (793 loc) · 35.3 KB
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     }
```

脆弱性の原因箇所

```
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
#
```

exploitコード実行例

デモ

■デモ動画を再生する

脆弱性報告

■ 任意コード実行が可能と判明した5件の脆弱性に関して、IPAやJPCERT/CCを通して開発者への報告を実施した

パッケージ名	届出	受理	開発者通知	パッチリリース	アドバイザリ	CVE	CVSSスコア
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

アドバイザリ(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

Versions

1 Total

Default Status: unknown

Affected

affected at prior to 1.01

Published:2024/08/28 Last 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.

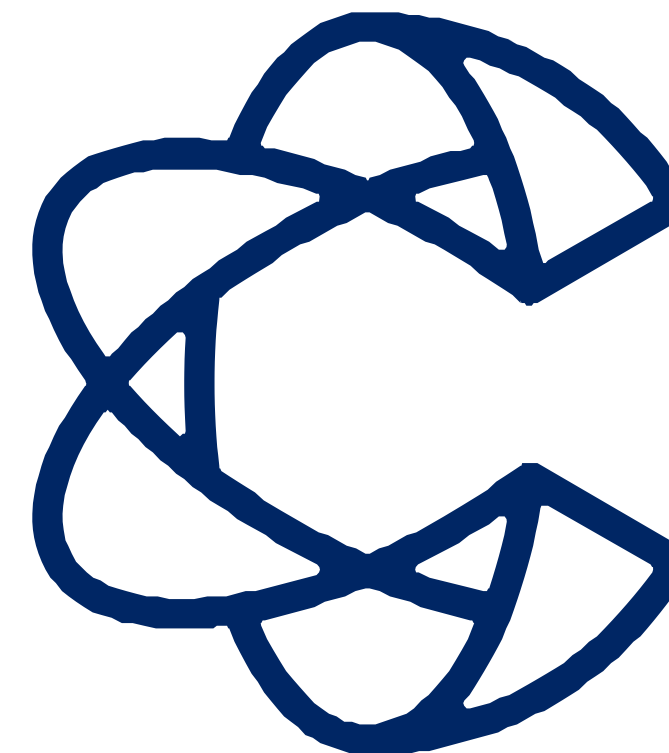
Credit

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

https://www.cve.org/CVERecord?id=CVE-2024-43700
https://jvn.jp/en/vu/JVNVU96498690/index.html

発表の内容

- 脆弱性とファジング
- PkgFuzz
- PkgFuzz Project
- クラッシュから脆弱性へ
- **今後の課題**
- まとめ



さらなる自動化に向けて

- PkgFuzz Projectでは一部の自動化阻害要因を解消したが、ファジング前後の両方の両方の段階において、まだ自動化できていない部分も存在する
- さらなる自動化に向け、次の課題について議論する
 - ハーネスの生成
 - 根本原因解析(RCA)

ハーネスの生成

- PkgFuzzでの課題：パッケージにテストが含まれていないものやプログラム実行時にファイルパスを引数として受け取らないものは対象外としており、これらに対応するにはハーネスの作成が必要となる
- 既存手法：
 - ハーネスの作成は通常は手動で行われていることが多いが、自動化するための手法やツールもいくつか提案・公開されている
 - LLMを活用したアプローチも有望だが、まだ様々な課題（複雑な仕様のAPIに対応できない、使用コスト等）があり、LLMとプログラム解析のハイブリッドが必要とされている^{*1}
- 今後：PkgFuzz Projectでも、自動ハーネス生成方法の適用により、より多くのパッケージに対する自動化を目指したい

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

根本原因解析 (RCA)

- PkgFuzzでの課題：攻撃可能性の調査過程の一つとして、クラッシュの根本原因箇所を特定する作業を行っているが、手動で行っているため時間がかかる
 - バグの原因箇所はクラッシュ箇所と必ずしも同じではないため
- 既存手法：
 - 一般的にはクラッシュ時の実行時情報をヒントにプログラムのコードを読んで手動で調査するが、自動化のための手法・ツールも提案されている
 - クラッシュ入力・非クラッシュ入力の実行時差に基づいた統計的なアプローチ（Aurora, VulnLoc等）
 - シンボリック実行に基づいたルールベースのアプローチ(Arcus)
 - 自動化ツールは、まだ実用上は精度やユーザビリティに課題を抱えている
- 方針：PkgFuzz Projectでは、既存ツールが抱える課題の改善を通じて、根本原因解析の自動化を目指す

RCAツールでの実行例

■ PkgFuzzが発見した脆弱性の根本原因を、既存ツール(Aurora, VulnLoc)が発見可能か？

- 原因箇所候補の順位一覧が出力される
- あらかじめ特定した正しい原因箇所が、一覧の何位に含まれているか確認（解析時間12h）

■ **sdop/CVE-2024-41881**【成功例】

- 正確に特定することができた（1位）

■ **assimp**/CVE-2024-40724【失敗例】

- 正確な特定はできなかった（300位以下）
- 実際の原因箇所が、クラッシュ時・非クラッシュ時の両方で実行されることが理由と考えられる

→ 成功するケースもあるが、失敗時に解析者の負担が増大するため、実用には精度やユーザビリティの改善が必要

```
[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)
...
```

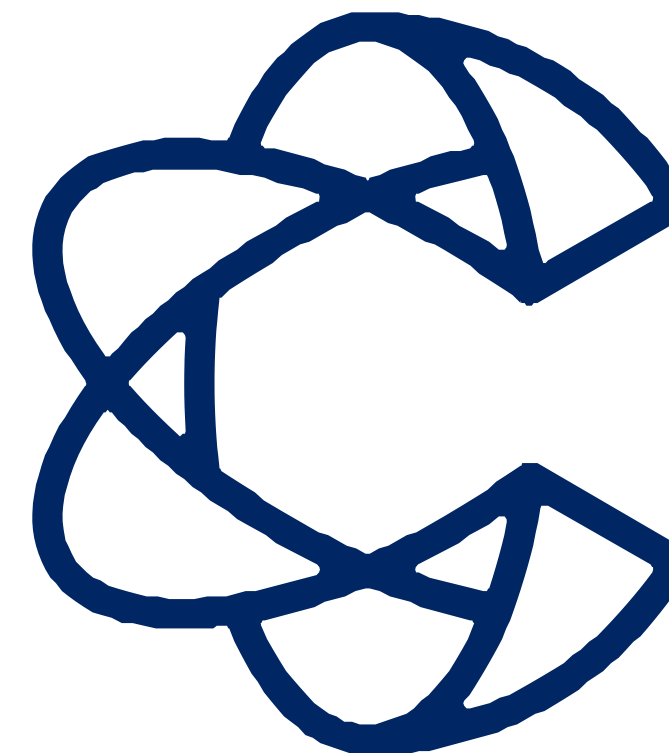
出力された根本原因候補(sdop)

```
src/read.c
@@ -3,7 +3,7 @@
*****/
4
5 /* Copyright (c) Philip Hazel, 2023 */
6 - /* Created in 2006; last modified: March 2023 */
7 + /* Created in 2006; last modified: June 2024 */
8
9 /* This module contains functions for reading a source file and parsing it into
a sequence of chained item blocks. Chapters and sections are numbered as we
@@ -93,8 +93,8 @@ for (param = new->p.param; param != NULL; param = param->next)
93 if (Ustrcmp(param->name, "revisionflag") == 0)
94 {
95     if (Ustrcmp(param->value, "changed") == 0) continue;
96 - (void)sprintf(CS buffer, "%s=%s:%s", param->name, param->value,
97 - new->name);
98 + (void)sprintf(CS buffer, "%s=%.*s:%s", param->name, 256 - 2*DBNAME_SIZE - 3,
99 + param->value, new->name);
100 }
101
102 /* See if this attribute is listed as supported unless the first item
```

実際の修正パッチ(sdop)

発表の内容

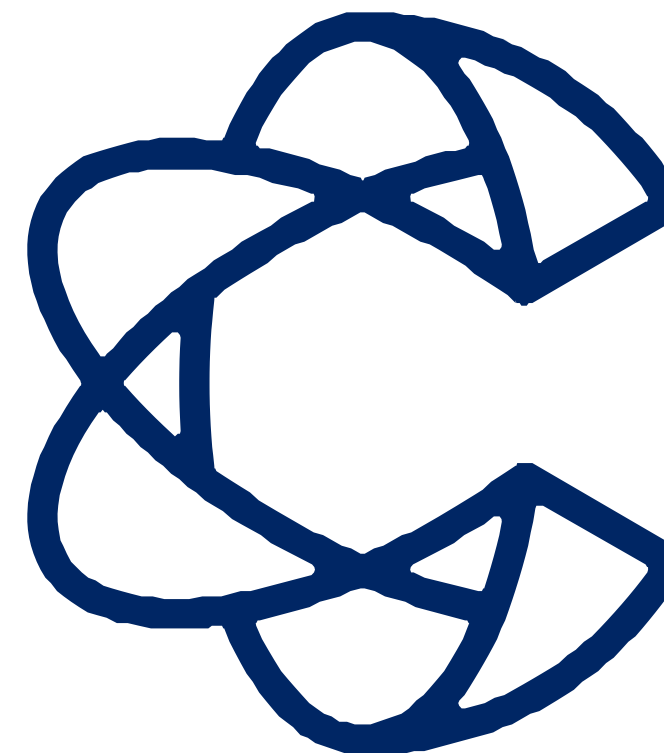
- 脆弱性とファジング
- PkgFuzz
- PkgFuzz Project
- クラッシュから脆弱性へ
- 今後の課題
- **まとめ**



まとめ

- OSSに対するもう一つのファジングプロジェクトとして、PkgFuzz Projectについて紹介した
 - ファジングワークフローを自動化するPkgFuzzにより、265のDebianパッケージに対して、64,658件のクラッシュ（トリアージ後1,186件）を自動的に検出し、5件のアドバイザリとCVEの発行につなげた
 - OSS-Fuzzと比較したところ、PkgFuzz Projectでは、人間がファジングを行った場合とほぼ同等の結果を自動的に（人間の介入なし）に実施できた。
 - OSS-Fuzzが対象としていないOSS(例えば小規模なOSS)に対してファジングテストを実行できた
 - OSS-Fuzzとの重複は12%=32/265
- PkgFuzz Projectのさらなる自動化に向けて、既存研究のサーベイを交えながら考察した。
- 引き続き、PkgFuzz Projectを進めていくことで、より多くのOSSの安全性を向上させ、安心なソフトウェア基盤の実現に寄与できると考える。

参考



ファジングの種類

種類	説明	ツール
ブラックボックス	検査対象の内部状態を考えない。ランダムテストと同義。	Peach、SPIKE、Sully
グラマーベース	ユーザが入力形式を指定する入力文法を提供する。	Peach、SPIKE、Sully
グレーボックス	検査対象の内部状態を粗く捉え、入力値の変化のヒントにする	AFL、AFL++、libFuzzer、Honggfuzz
ホワイトボックス	検査対象の内部状態細かく捉えながら解析を進める。シンボリック実行と同義。	SAGE、KLEE、S2E
ハイブリッド	グレーボックスとホワイトボックスを組み合わせたもの。	Driller、Qsym

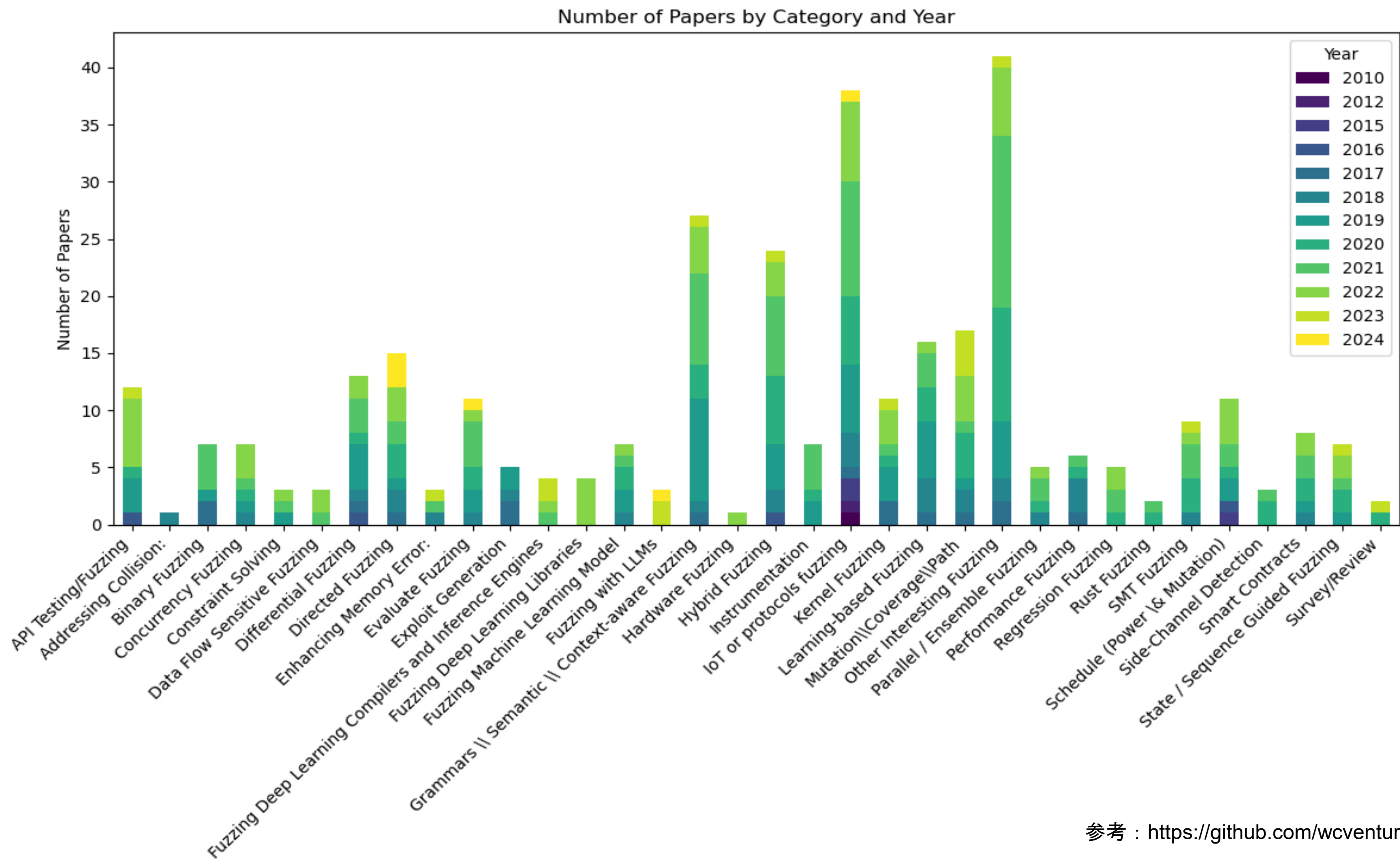
単純



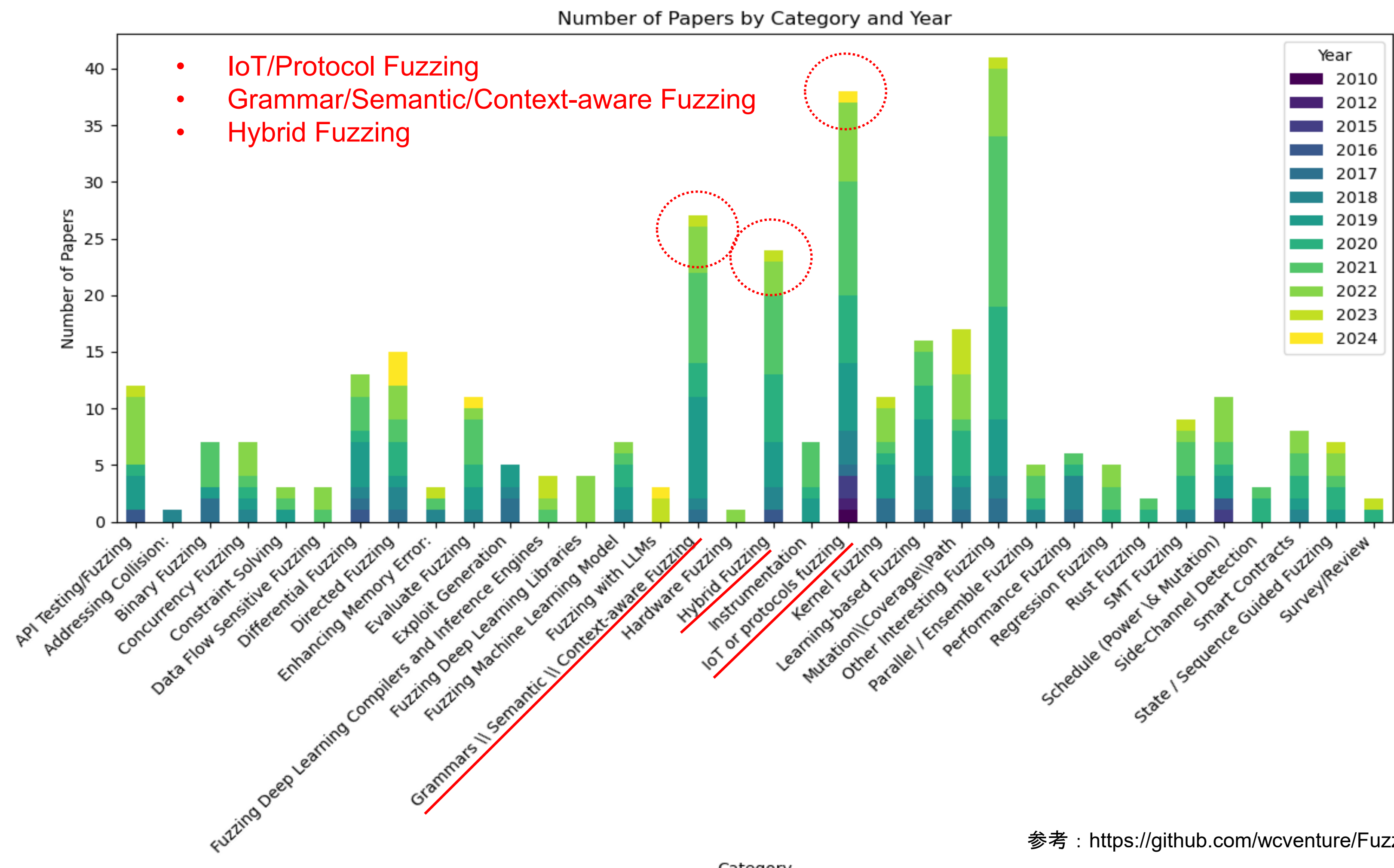
複雑

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

ファジングの学術的研究



ファジングの学術的研究：研究が多い分野



ファジングの学術的研究：最近の流行り

