GreyBox Fuzzing

ICT RISK ASSESSMENT,

LORENZO BELLOMO

Outline

1. Introduction: what is GreyBox Fuzzing
2. AFL, American Fuzzy Lop
3. AFLGo, Directed GreyBox Fuzzer
4. AFLSmart, Smart GreyBox Fuzzer
5. AFLSmart demo

Towards GreyBox Fuzzing

WHITE BOX FUZZING

Analyses the program structure

- Generally effective
- Good code coverage

But:

- Requires heavy machinery (solving optimization problems)
- This time is not spent fuzzing

BLACK BOX FUZZING

No assumptions on the source code

- Always usable
- Fast in generating new inputs

But:

- It is harder to create valuable inputs
- Hard to get feedback from the fuzzing session

GreyBox Fuzzing

Doesn't use program analysis, but *instrumentation* (takes the best of white/black box fuzzing)

The fuzzing session is analysed in order to:

- *Monitor* the explored paths (basic blocks transitions)
- Favor some paths with respect to others (*fuzzing policy*)
- Mutate *valuable* inputs

Without paying the cost of optimization problems

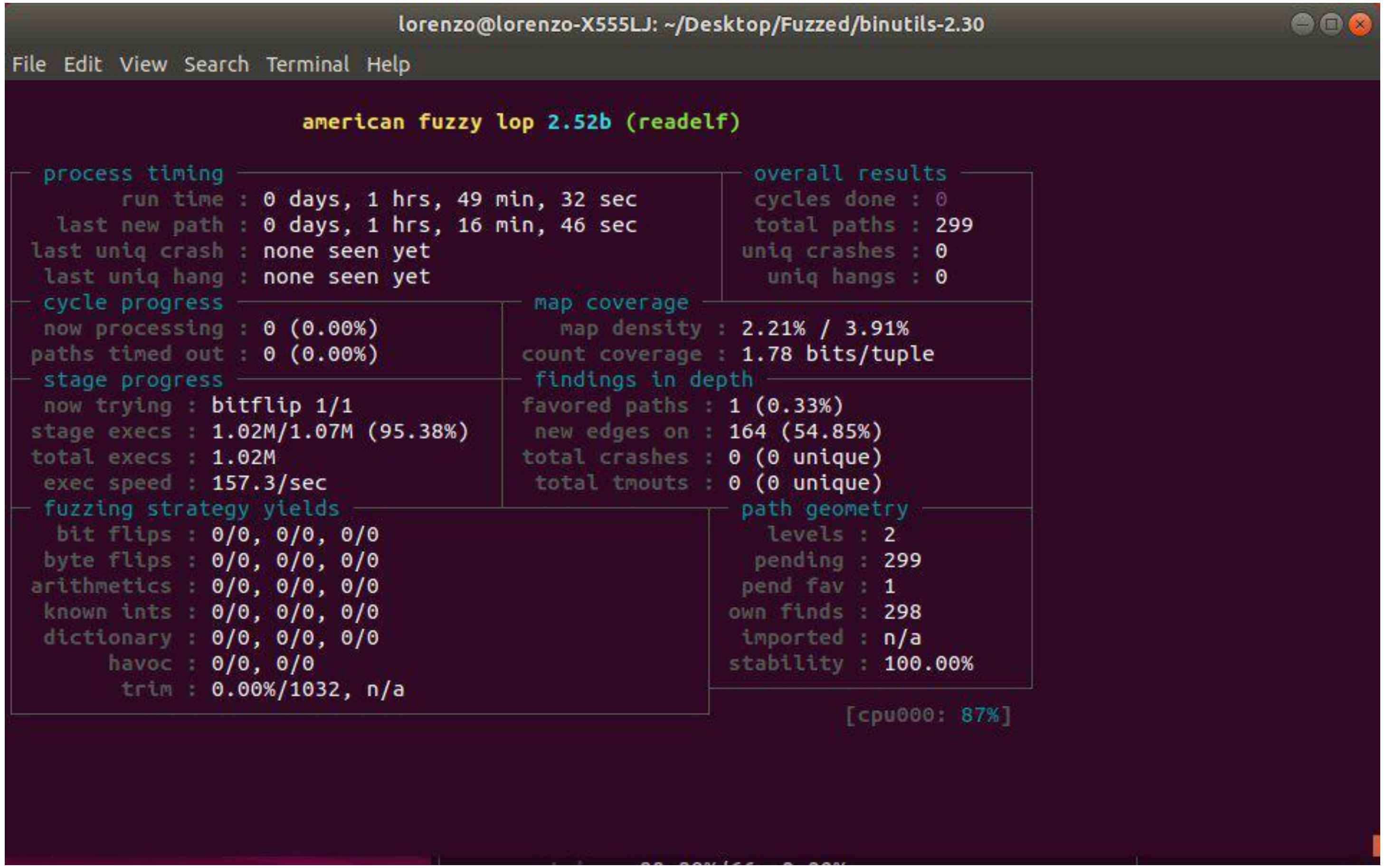
American Fuzzy Lop (AFL)

Most known *Greybox fuzzer*

Released in 2015

Open Source code available (Github link)

Highly used and optimized



Main features of AFL

Provides a a *wrapper* to compile directly setting all the things needed for instrumentation

```
$ CC=/path/to/afl/afl-gcc ./configure
```

Also provides wrappers for g++ and clang

Mutates the provided inputs with *bit-wise operations*

Provides many ways to improve fuzzing:

- *Dictionaries*: allow to specify abstraction over file formats
- *Parallel fuzzing*: Every instace of afl-fuzz takes one core
- …

AFL basic loop

1. Load user-supplied initial test cases into the queue
2. Take next input file from the queue
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program
4. Repeatedly *mutate* the file using a balanced and well-researched variety of traditional fuzzing strategies
5. If any of the generated mutations resulted in a *new state* transition recorded by the instrumentation, *add mutated output* as a new entry in the queue
6. Go to 2.

AFL, how it works

It uses *genetic algorithm* to maximize coverage

It assigns most *energy* to inputs that:

- Got in *rare paths*
- Executed for a *small* amount of time

It also allows to make *BlackBox Fuzzing* (at the expense of performance) through QEMU mode

Optimized heavily to avoid using `execve()` by exploiting fork and copyOnWrite (More informations)

AFL extensibility

AFL fuzzes in a very specific way:

- Optimization in *performance*, at the expense of discarding potentially *buggy but slow* inputs
- Very *easy to use*, at the expense of *flexibility*
- Maximizes *coverage*, at the expense of *interesting paths*

Extend/Modify the source to change objectives:

- What if I wanted to *direct* a program to specific lines of code?
- What if I wanted to target a specific *file format* application (JPEG, PDF…)

Two AFL extensions

AFLSmart

Aims at creating *valid* seed files

Modifies the way in which AFL builds new *seed* files

Discards some *performance* with respect to AFL

AFLGo

Aims at *directing* the fuzzing procedure towards specific code

Modifies the *energy management* process of AFL

Discards the *coverage* part of AFL

AFLGo

The main goal is to *direct* the input towards *critical code*, useful for:

- *patch testing*: directing the fuzzing process to the newly changed code
- *crash reproduction*: usually related with anonymous clients reports after a crash
- *static analysis report verification*
- *information flow detection*: to reach private information sources and sinks that make them public

Github link

Algorithm 1 Greybox Fuzzing

```
Input: Seed Inputs S
repeat
    s = CHOOSENEXT(S)
    p = ASSIGNENERGY(s)                         // Our Modifications
    for i from 1 to p do
        s′ = MUTATE_INPUT(s)
        if t′ crashes then
            add s′ to S✗
        else if ISINTERESTING(s′) then
            add s′ to S
        end if
    end for
until timeout reached or abort-signal
Output: Crashing Inputs S✗
```

AFLGo main loop

The modification is in the `assignEnergy(seed)` procedure

This is in order to favor inputs that go closer to the target

Energy is related to the *simulated annealing* algorithm

HeartBleed (1)

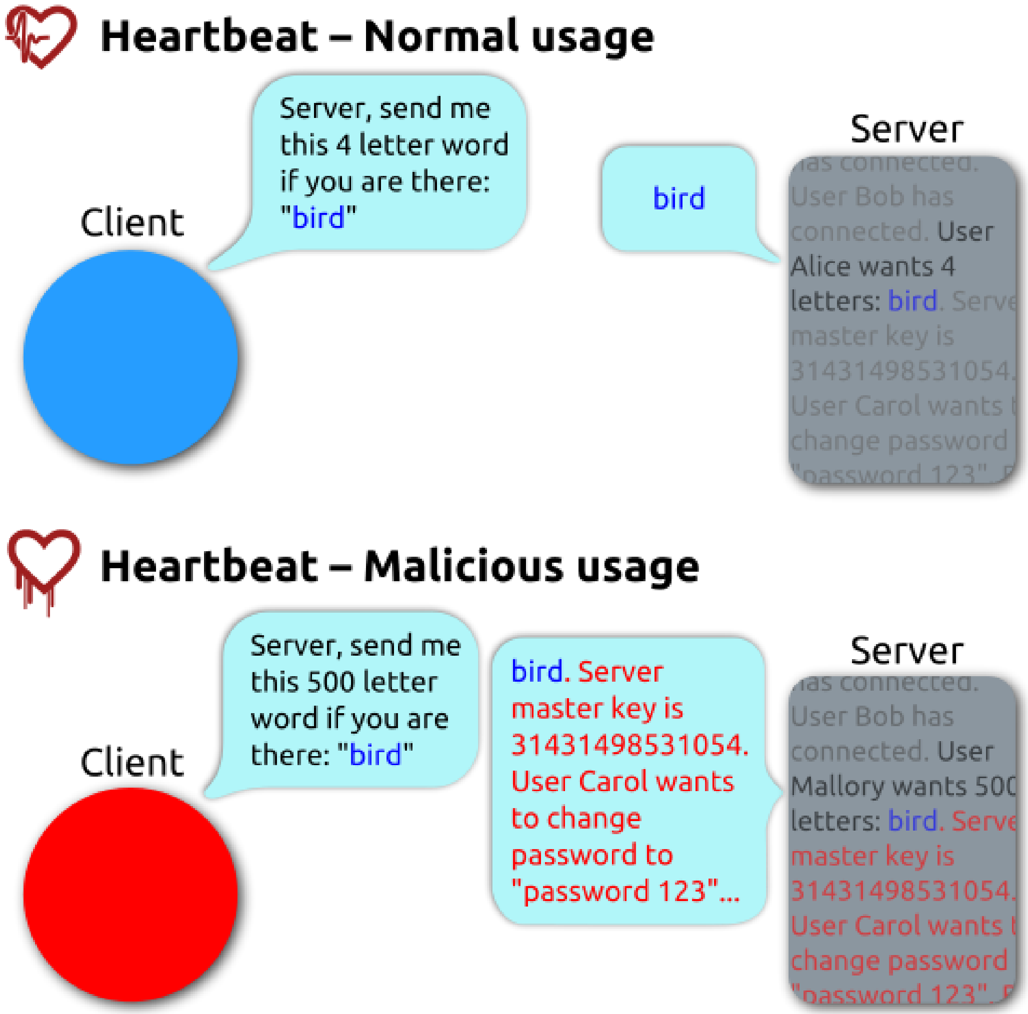
We take as example vulnerability Heartbleed (*CVE-2014-0160, 2012-2014*)

Known *OpenSSL* client bug for the Heartbeat extension

Actually a *buffer over-read*, due to incorrect bound checks

```
1455 + /* Read type and payload length first */
1456 + hbtype = *p++;
1457 + n2s(p, payload);
1458 + pl = p;
...
1465 + if (hbtype == TLS1_HB_REQUEST) {
1477 +     /* Enter response type, length and copy payload */
1478 +     *bp++ = TLS1_HB_RESPONSE;
1479 +     s2n(payload, bp);
1480 +     memcpy(bp, pl, payload);
```

Figure 1: Commit introducing Heartbleed: After reading the `payload` from the incoming message `p` (1455-8), it copies `payload` many bytes from the incoming to the outgoing message. If `payload` is set to 64kb and the incoming message is one byte long, the sender reveals up to ~64kb of private data.



HeartBleed (2)

This bug was added on New Year's Eve 2011 by commit 4817504d

OpenSSL code is over 500000 lines of code, so we shouldn't fuzz the whole application for a small patch/commit

The other method used to obtain *directed execution* is through *symbolic execution*

Identify valuable inputs

I need to assign a *score* to the seeds, based on the *average distance* to the *target set*.

$$d(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|}$$

- s is the *seed*
- T_b is the *target set*
- $d_b(m, T_b)$ is the *average distance* from a *basic block* and the *target set*
- $\xi(s)$ is the execution trace of seed s. It basically contains all the *exercised basic blocks*

The computation of $d(s, T_b)$ is the only runtime *overhead*, all the rest is computed *statically*

Merge AFL and AFLGo metrics

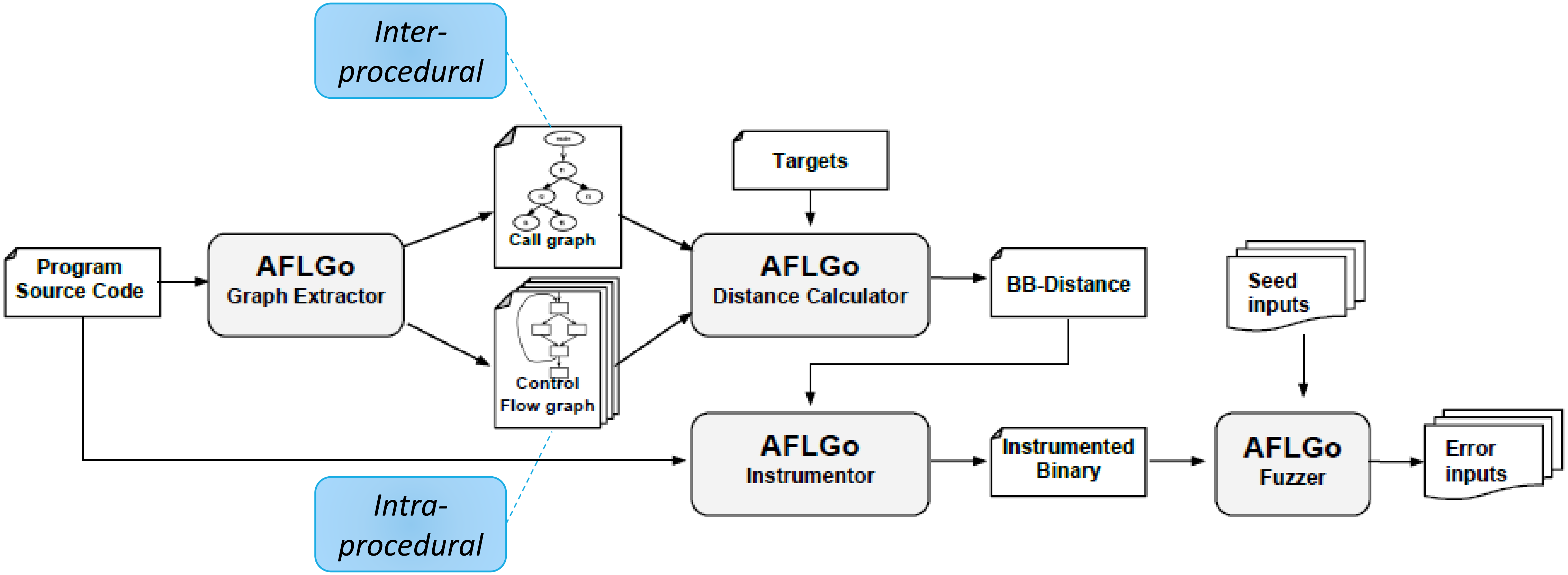
AFL and AFLGo metrics are merged with simulated annealing

The process is regulated by the (global) *temperature*

$$p(s, T_b) = \left(1 - \check{d}(s, T_b)\right) * \left(1 - T_{exp}\right) + 0.5T_{exp}$$

Exploration (first phase): more energy is assigned to futher seeds

Exploitation(second phase): more energy is assigned to closer ones



AFLGo architecture

Integration with OSS-Fuzz

OSS-Fuzz (*Google*) is a *continuous testing platform*

It fuzzes new releases of *registered* products to find security vulnerability

Fuzzes also huge projects (Firefox, CPython, OpenSSL …)

Open source code (Github link)

OSS-Fuzz fuzzes the latest release, but *it is not directed* towards the last commit

AFLGo provides an integration with OSS-Fuzz to actually *fuzz patches*

AFLSmart

AFLSmart is built on two fuzzers:

- AFL, for the most part
- PEACH fuzzer (blackbox), mainly for the file format specifications

The important part is the *file format specification*

For each format type (MP3, JPEG, WAV…) a *virtual structure* must be built

Github link

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Corpus S

repeat
  $s = \text{CHOOSENEXT}(S)$ *// Search Strategy*
  $p = \text{ASSIGNENERGY}(s)$ *// Power Schedule*
  for i from 1 to p **do**
    $s' = \text{MUTATE\_INPUT}(s)$
    if s' crashes **then**
      add s' to $S_{\times}$
    else if $\text{ISINTERESTING}(s')$ **then**
      add s' to S
    end if
  end for
until *timeout* reached or *abort*-signal

Output: Crashing Inputs $S_{\times}$

AFLSmart main loop

The modification is in the `MUTATE_INPUT(seed)` procedure

It uses bitwise operations, but makes them *smart* by respecting the file format specification

Allowing to pass the *parsing* phase

SOI	0xFF, 0xD8	*none*	Start Of Image
SOF0	0xFF, 0xC0	*variable size*	Start Of Frame (baseline DCT)
SOF2	0xFF, 0xC2	*variable size*	Start Of Frame (progressive DCT)
DHT	0xFF, 0xC4	*variable size*	Define Huffman Table(s)
DQT	0xFF, 0xDB	*variable size*	Define Quantization Table(s)
DRI	0xFF, 0xDD	4 bytes	Define Restart Interval
SOS	0xFF, 0xDA	*variable*	Start Of Scan

File format, JPEG example

From Wikipedia: “*A JPEG image consists of a sequence of segments, each beginning with a marker, each of which begins with a 0xFF byte, followed by a byte indicating what kind of marker it is.*“

In the picture, some common markers are shown

The work of defining the specifications is done once, and then *re-used* for the same format

```xml
<DataModel name="Jpeg">
    <Number name="SOI" value="FF D8" size="16"/>
    <Choice name="Segments" maxOccurs="10000">
        <Block name="AppSeg" ref="APPSegment"/>
        <Block name="SofSeg" ref="SOFSegment"/>
        <Block name="DhtSeg" ref="DHTSegment"/>
        <Block name="DqtSeg" ref="DQTSegment"/>
        <Block name="DriSeg" ref="DRISegment"/>
        <Block name="SosSeg" ref="SOSSegment"/>
        <Block name="RstSeg" ref="RESTARTMarker"/>
    </Choice>
    <Blob name="ScanData"/>
```

```xml
<DataModel name="SOFSegment" ref="MarkerSegmentWithPayload">
    <Choice name="Marker">
        <Number name="Marker0" value="FF C0" size="16" token="true"/>
        <Number name="Marker1" value="FF C1" size="16" token="true"/>
        <Number name="Marker2" value="FF C2" size="16" token="true"/>
    </Choice>
</DataModel>

<DataModel name="DHTSegment" ref="MarkerSegmentWithPayload">
    <Number name="Marker" value="FF C4" size="16" token="true"/>
</DataModel>

<DataModel name="DQTSegment" ref="MarkerSegmentWithPayload">
    <Number name="Marker" value="FF DB" size="16" token="true"/>
</DataModel>
```

JPEG file specification (PEACH)

File specifications vs AFL Dictionaries

AFL provides already *dictionaries* for abstracting the file structure

From AFL documentation: “…*afl-fuzz provides a way to seed the fuzzing process with an optional dictionary of language keywords, magic headers, …and use that to reconstruct the underlying grammar on the go*“

The problem is that the following modifications are bitwise, while we want to add/remove chunks (consistently w.r.t. the format)

Mutating seeds

AFLSmart uses two types of *mutations*

- *Low Level*: Similar to the ones of AFL, bitwise operations like shift, removing/adding random bits
- *High Level*: Addition/removal of whole chunks of data

smart operations: Deletion, Addition, Splicing

More energy is assigned to more valid seeds

$$p_v(s) = \begin{cases} 2p(s) \; if \; v(s) \geq 50\% \; and \; p(s) \leq U/2 \\ p(s) & if \; v(s) < 50\% \\ U & otherwise \end{cases}$$

Solving Scalability issues

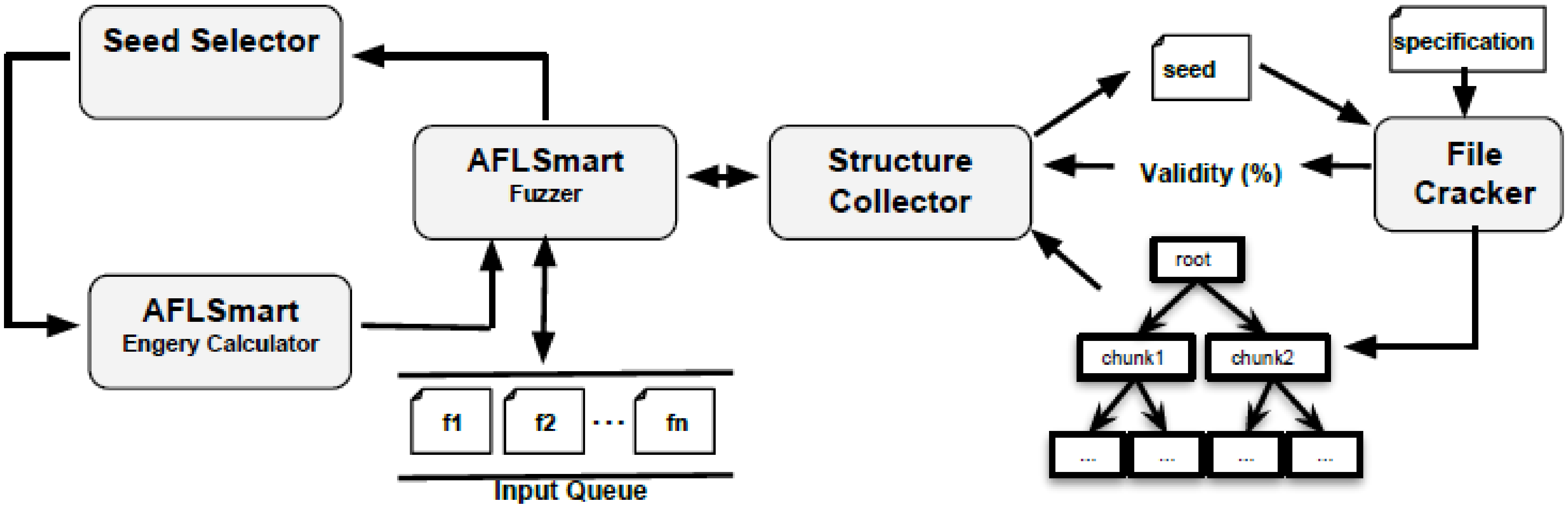
Anytime AFLSmart checks the *virtual structure*, it needs 2/3 seconds

This process can’t be done for all the input seeds

So we use a *probability* to check the virtual structure, which increases more and more while no new paths are discovered

Basically AFLSmart is exploited mostly when AFL *struggles*

$$prob_{virtual}(s) = \min({}^{t}/_{\epsilon}, 1)$$



AFLSmart architecture

AFLSmart, Test case

AFLSmart is built on top of AFL, so it provides all the options that AFL provides, plus:

-w: *input model type* (peach only at the moment)

-g: *input model file* (path to the model of the file)

-h: *stacking mutations mode* (mixes normal and higher-order mutations)

-H: limit the number of *higher order mutations* for each input

AFLSmart, fuzzing WavPack

The first test was made against a known *vulnerable commit* of *WavPack*, an audio file compressor.

The commit is the 0a72951, and the goal is to reproduce *CVE-2018-10536*

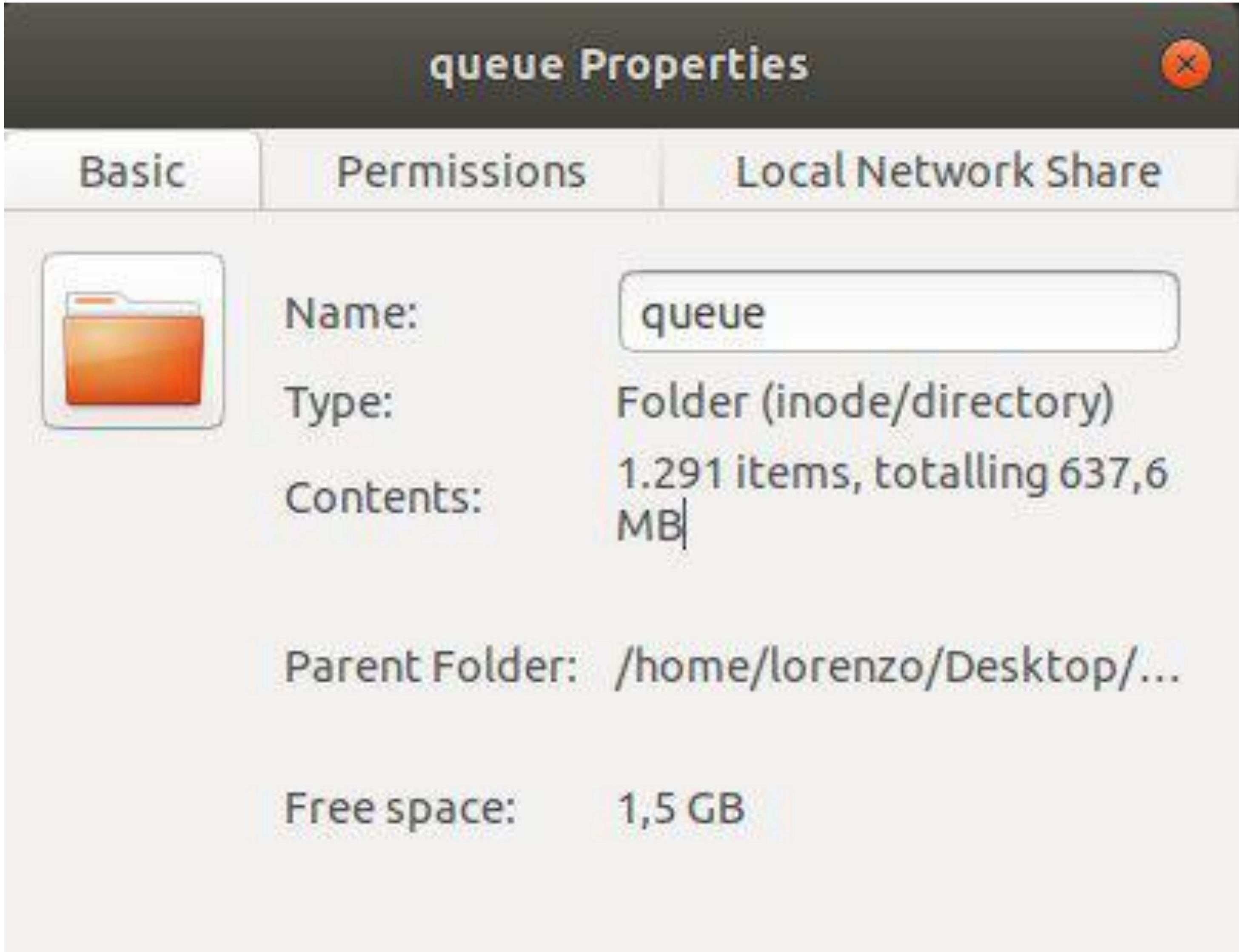
"*The WAV parser component contains a vulnerability that allows writing to memory because ParseRiffHeaderConfig in riff.c does not reject multiple format chunks*"

We're going to learn all the fuzzing steps with WavPack, but then concentrate on another binary (*readelf*)

How to fuzz

```
$AFLSMART/afl-fuzz -m none -h -d -i
$AFLSMART/testcases/aflsmart/wav -o
out -w peach –g
$AFLSMART/input_models/wav.xml –x
$AFLSMART/dictionaries/wav.dict
cli/wavpack -y @@ -o out
```

- $AFLSMART was set as the path to the AFLSmart directory
- There are 2 [-o], one for the fuzzer and one for the program
- @@ is a placeholder for the files given to the program
- The fuzzed binary is cli/wavpack
- wav.dict is the standard AFL dictionary, wav.xml is the newly added format specification



Post runs details

The *queue* folder contains the history of the *fuzzing phase.*

It gets filled with “fuzzier and fuzzier” inputs

The longer the run goes, the more items get in this folder

It grows very fast in size (1GB reached in around 3 hrs)

Crashes folder

It contains the files that produced a *crash*

The name assigned is in this form:

- *id*: increasing integer identifying crashes
- *sig*: the signal id that caused the crash (11 is SIGSEGV, 06 is SIGABRT…)
- *src*: non crashing source file id (from queue folder)
- *op*: the operation that transformed the file from a normal one to a crash

Analyze runs output

To recreate the crashes, just call the program in the same way that was specified in the fuzzer

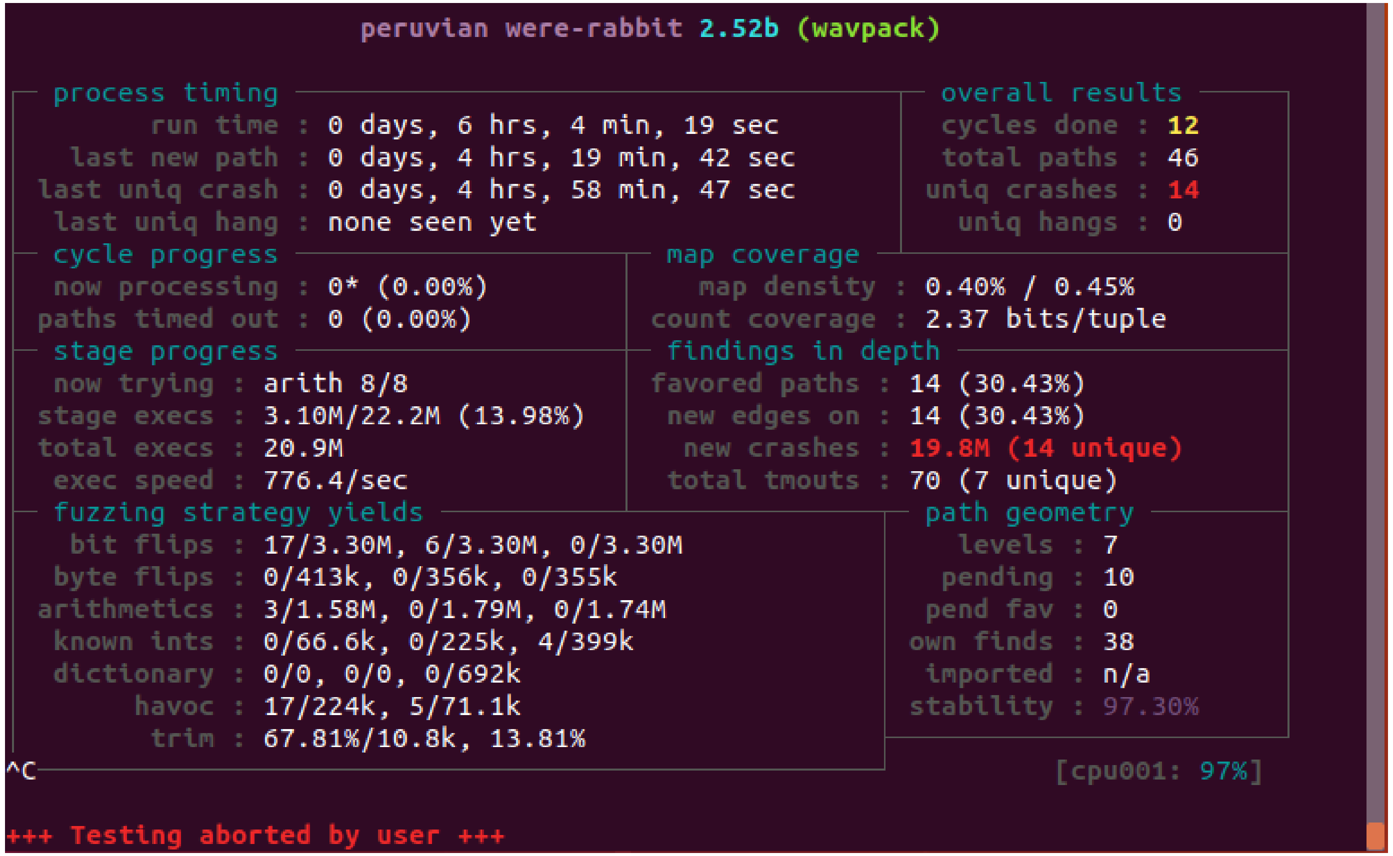
It was: `cli/wavpack -y @@ -o out`

Analyze it with a *debugger* (Valgrind), substitute @@ with the file name

Redirect *stdout* and *stderr* to a file

```
 valgrind cli/wavpack –y
path/to/crashing_file -o
dump_folder >> valgrindDump.txt
2>&1
```

```
22380== Command: cli/wavpack out/crashes/id:000000,sig:11,src:000001,op:havoc,rep:8.wav -o /home/lorenzo/Desktop/
22380==

AVPACK  Hybrid Lossless Audio Compressor  Linux Version 5.1.0
opyright (c) 1998 - 2017 David Bryant.  All Rights Reserved.

eating id:000000,sig:11,src:000001,op:havoc,rep:8.wv,==22380== Argument 'size' of function malloc has a fishy (possibly negative) value: -65536
22380==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
22380==    by 0x150C7F: ParseRiffHeaderConfig (riff.c:289)
22380==    by 0x148E8D: pack_file (wavpack.c:1776)
22380==    by 0x10EF02: main (wavpack.c:1272)
22380==
22380== Invalid write of size 1
22380==    at 0x4C371BC: mempcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
22380==    by 0x52656C7: _IO_file_xsgetn (fileops.c:1326)
22380==    by 0x52593C0: fread (iofread.c:38)
22380==    by 0x16A966: fread (stdio2.h:294)
22380==    by 0x16A966: DoReadFile (utils.c:618)
22380==    by 0x150D18: ParseRiffHeaderConfig (riff.c:296)
22380==    by 0x148E8D: pack_file (wavpack.c:1776)
22380==    by 0x10EF02: main (wavpack.c:1272)
22380==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
22380==
22380==
22380== Process terminating with default action of signal 11 (SIGSEGV)
22380==  Access not within mapped region at address 0x0
22380==    at 0x4C371BC: mempcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
```



Peruvian were-rabbit

At the end of an AFL run, some *crashes* have been produced

AFL provides a way to start another run, starting from those *crashing inputs*

It *ignores* coverage,

It *stresses* buggy inputs from the crash folder

It is called *Peruvian were-rabbit*

Another step, exploitable

Idea: Use *GDB* to analyze all the runs automatically

Integrate the result with the GDB extension *exploitable,* which analyzes the crash and tells if it might be a security vulnerability (Github link).

It maps the crash into those categories:

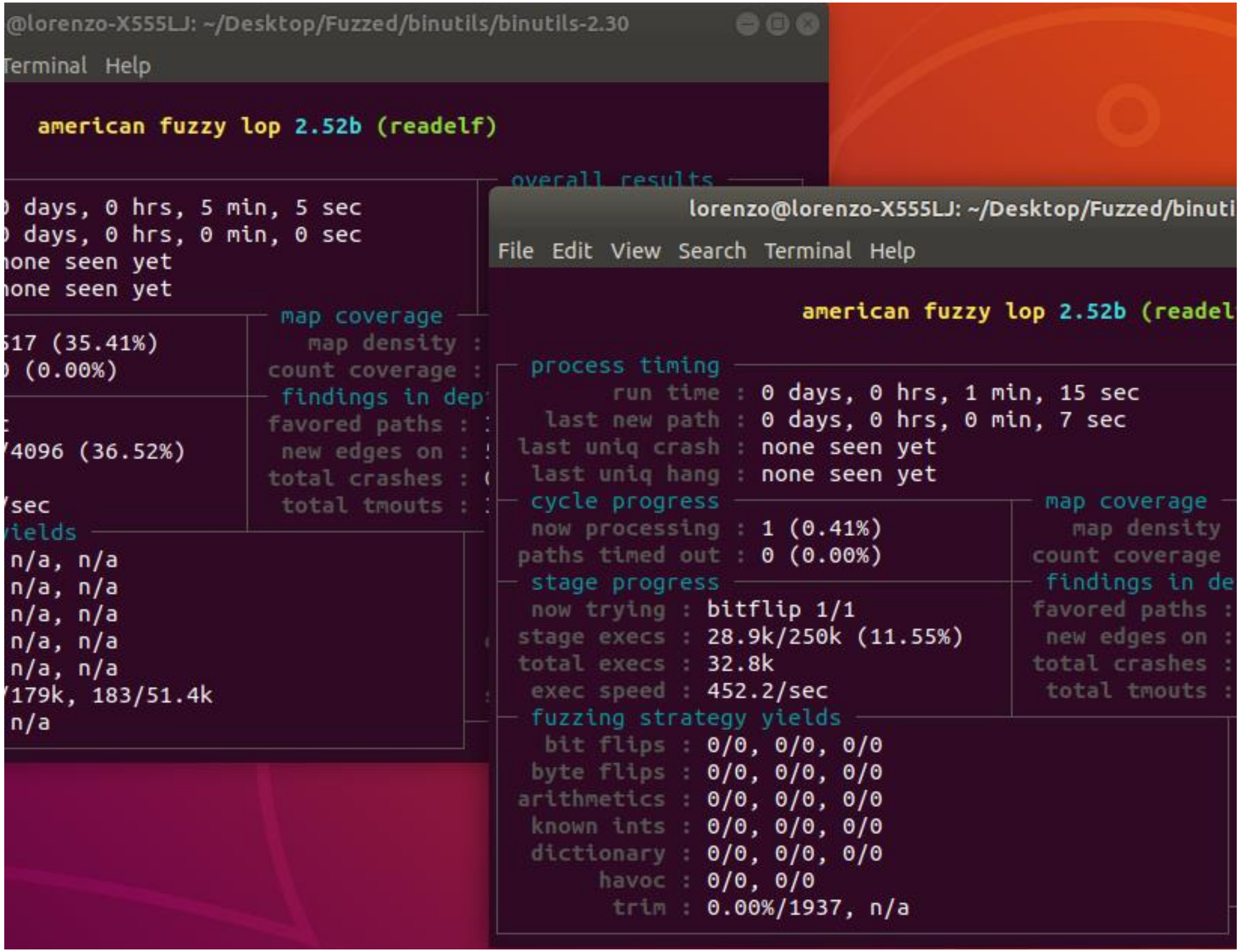
- *Exploitable*
- *Probably exploitable*
- *Probably not exploitable*
- *Unknown*

The script to automate it

```
"" > ~/Desktop/gdbExplOut.txt

for file in ~/Desktop/Fuzzed/data/binutils/
peruvian/crashes/*
do
  gdb -x ~/Desktop/gdbScript --args ~/Desktop/
Fuzzed/binutils/binutils-2.30 $file >> ~/Desktop/
gdbExplOut.txt &
  pid=$!
  sleep 4
  kill -TSTP pid # ctrl-z
done
```

gdbScript:

```
source
~/path/to/exploitable.py

run

exploitable
```



Fuzzing readelf

Having all the ingredients, now we fuzz something *huge*

readelf from *binutils*

At the beginning both with AFL and AFLSmart

Later only with the most performing one (*AFLSmart*)

After one hour

```
   map density : 1.69% / 13.43%
count coverage : 3.08 bits/tuple
 findings in depth
favored paths : 1179 (20.68%)
 new edges on : 1987 (34.85%)
total crashes : 100 (21 unique)
 total tmouts : 85 (25 unique)
```

```
   map density : 1.94% / 6.47%
count coverage : 2.47 bits/tuple
 findings in depth
favored paths : 15 (1.47%)
 new edges on : 466 (45.82%)
total crashes : 0 (0 unique)
 total tmouts : 31 (11 unique)
```

```
  overall results
 cycles done : 0
 total paths : 5807
uniq crashes : 21
  uniq hangs : 0
```

```
, 38 sec
, 30 sec
```

```
  overall results
 cycles done : 0
 total paths : 1017
uniq crashes : 0
  uniq hangs : 0
```

Some data

After 1 day 128 unique crashes were found

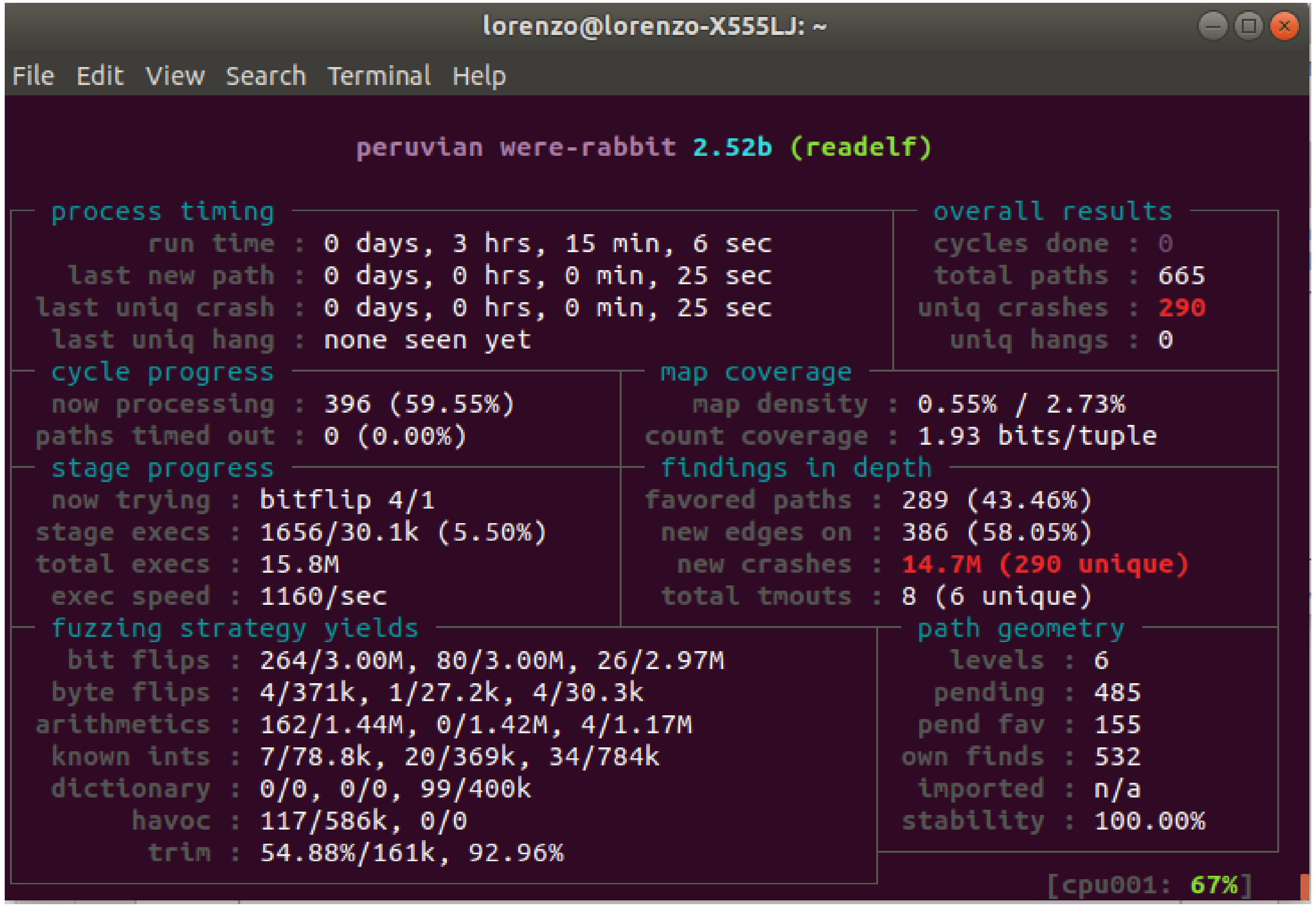
Then the process slowed down

133 crashes after 2 days and 7 hours

It was also possible to see the AFLSmart *virtual structure* being built:

- Checking runtime stats, the *performance drops* every now and then from around 1000 exec/sec to 10
- This lasts 1-2 seconds

Stressing crashes



Now it is time to stress the *crash* folder (133 items) with the *peruvian were-rabbit* mode

In 3 hours already more than *doubled* the amount of crashes

Notice:

- Low number of *explored paths*
- Huge number of *crashes*

Analyzing the final output

Using the GDB+exploitable script built before for WavPack

All the crashes (both pre and post peruvian mode) have the same nature (around 1000 unique crashes)

All classified as like follows:

"*Exploitability Classification: UNKNOWN*

Explanation: The target is stopped on a SIGABRT. SIGABRTs are often generated by libc and compiled check-code to indicate potentially exploitable conditions. Unfortunately this command does not yet further analyze these crashes."

```c
/* Return a pointer to section NAME, or NULL if no such section exists.  */

static Elf_Internal_Shdr *
find_section (Filedata * filedata, const char * name)
{
  unsigned int i;

  assert (filedata->section_headers != NULL);

  for (i = 0; i < filedata->file_header.e_shnum; i++)
    if (streq (SECTION_NAME (filedata->section_headers + i), name))
      return filedata->section_headers + i;

  return NULL;
}
```

```
Assertion `filedata->sec...   1 of 133

detector
and GNU GPL'd, by Julian Seward et al.
 LibVEX; rerun with -h for copyright info
sktop/Fuzzed/binutils/binutils-2.30/binutils/readelf -a /home/lorenzo/Desktop/Fuzzed/d

n: Assertion `filedata->section_headers != NULL' failed.
```

The bug

A reachable assertion

Probably nothing exploitable

Just a *bug*, not a security one

All the unique crashes found are different executions reaching this assertion

Something more interesting

In more than 3 days of fuzzing readelf, nothing interesting was found

Let's roll back to the vulnerable WavPack (which was fuzzed for a few hours)

The interesting vulnerability (discussed in the paper) was not reached

The one that was found is marked as potentially exploitable, but it is actually a NULL pointer dereference

```
Invalid write of size 1
   at 0x4C371BC: mempcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x52656C7: _IO_file_xsgetn (fileops.c:1326)
   by 0x52593C0: fread (iofread.c:38)
   by 0x16A966: fread (stdio2.h:294)
   by 0x16A966: DoReadFile (utils.c:618)
   by 0x150D18: ParseRiffHeaderConfig (riff.c:296)
   by 0x148E8D: pack_file (wavpack.c:1776)
   by 0x10EF02: main (wavpack.c:1272)
 Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

```
Program received signal SIGSEGV, Segmentation fault.
__memmove_avx_unaligned_erms ()
    at ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:293
Description: Access violation near NULL on destination operand
Short description: DestAvNearNull (15/22)
Hash: f967d4d903a9f2677aaf9b67a8723b20.649a302fea002e5fea14231c3a18078f
Exploitability Classification: PROBABLY_EXPLOITABLE
Explanation: The target crashed on an access violation at an address matching the destination
operand of the instruction. This likely indicates a write access violation, which means the
attacker may control write address and/or value. However, it there is a chance it could be a
NULL dereference.
```

The bug

GDB (below) recognizes a possible write access violation

But it also might be a NULL pointer dereference

Valgrind (up) confirms it is not a security issue (address 0x0)