GameSS

EDUCATIONAL MATERIAL IN CYBER SECURITY

WHO IS THE MATERIAL FOR?

This module targets software engineers interested in increasing confidence that existing software projects are free of bugs by means of automated software analysis.

The module assumes some programming background, though not necessarily systems programming (some basic C will be used in the course). A general computer science background is beneficial, though not required.



WHO MADE THIS MATERIAL?

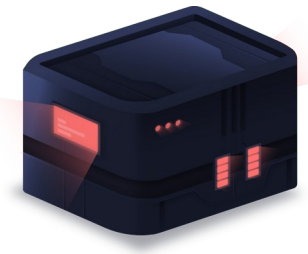
Alessandro Bruni

Associate Professor

IT-University of Copenhagen

WHAT ARE THE MAIN TAKEAWAYS FOR THIS CONTENT?

- Fuzzing and symbolic execution are techniques that help discovering software bugs and potential vulnerabilities.
- Through the use of **fuzzing**, one can find bugs in software by randomizing the input data. This is especially useful in checking programs that process untrusted data, and WEB APIs.
- Through the use of **symbolic execution**, one can find bugs in software by searching for specific paths of code execution that elicit some undesired behaviuor. Symbolic execution works well as a white-box testing technique, where both the program and the property are given to the user.



WHAT IS FUZZING

WHAT IS FUZZING?

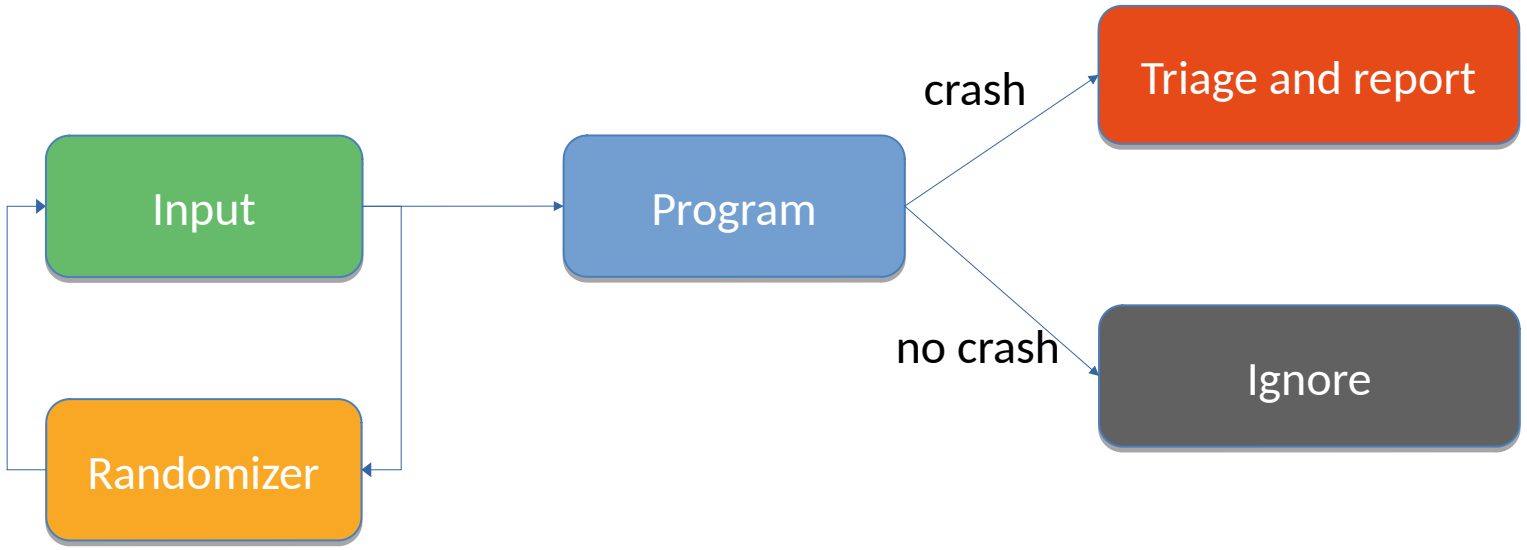
QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

Fuzzing is (almost like that)

Fuzzing is a technique to elicit bugs in software by means of running the program on a set of randomized inputs.

Compared to testing, where humans have to construct valid tests, inputs are produced procedurally.

HOW FUZZING WORKS



Fuzzing a program consists in running the program with random input and observing its behaviour: if the program crashes, then the input is saved for future inspection, and discarded otherwise. New inputs are generated by a randomizer that mutates an initial set of inputs to improve the chances of covering the state space of the program.

WHAT IS FUZZING GOOD FOR?

- Often used for file formats or networking protocols
- Easy to set up
- Almost no compute requirements for simple programs
- Generating/mutating inputs for a program
  - Large set of valid inputs (corpus) mixed together and corrupted
  - Read a specification for a protocol, design something to make valid traffic
- Goal of creating an exceptional condition
  - Program crash
  - Unique errors
  - Hangs

LIMITATIONS OF FUZZYING

- Generating all valid inputs is nearly impossible
- Custom protocol extensions hard to know about without reversing
- Non-crashing bugs typically not found
  - Arbitrary file read
  - SQL injection
  - Magical privilege escalation
- Hard to interpret progress
  - No crashes? Are there no bugs? Is your fuzzer not working?
  - Loads of crashes? Maybe it's working? Maybe you're finding 1% of the bugs?
- Most off the shelf tooling requires source

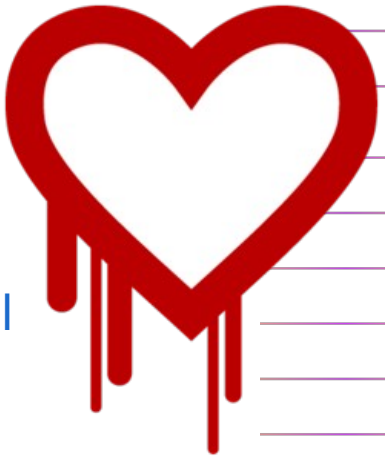
HARNESSING

- The tooling used to observe program behavior
  - Debugger watching for crashes
  - Code coverage instrumentation by a compiler
  - Emulators/hypervisors to observe a whole system
- What are they looking for?
  - Crashes
  - Code coverage
  - Unexpected program states
  - Error messages
  - Information leakage

SUCCESS STORIES

- **Google Project Zero** used fuzzing to uncover vulnerabilities in a range of open source software
  https://google.github.io/oss-fuzz/
- Fuzzing had a significant impact in finding the **Heartbleed** vulnerability in OpenSSL
  https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html
- Fuzzing was used to discover the **Microsoft SMB vulnerability** that was later exploited by EternalBlue and NotPetya

  https://msrc.microsoft.com/blog/2017/07/englishmansdentist-exploit-analysis/



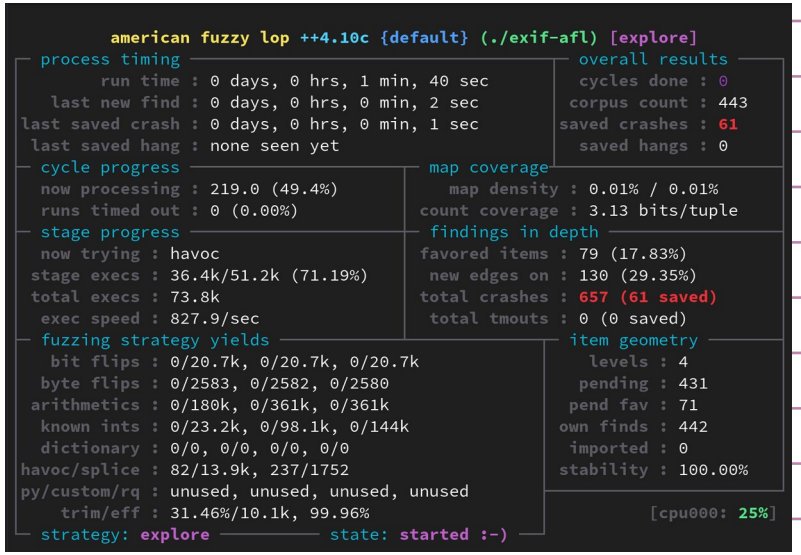
EXAMPLES

MY FIRST FUZZER: JUST INVOKE THE PROGRAM!

- Usually the right place to start
- Write some tool that generates a mutated file
- Run the program to parse the file
- What if we get a crash?
  - Ehh… just attach GDB or WinDbg or enable core dumps :D
- Reproducibility can be a huge issue
- Can be impossible to scale as the program can only have one instance
- Program startup times can be long (seconds to open up Word)

AFL++

- The gold standard
- Coverage guided fuzzing
- Requires source code
- Relies on having source in standard configuration
- Can use QEMU for coverage



LIBFUZZER



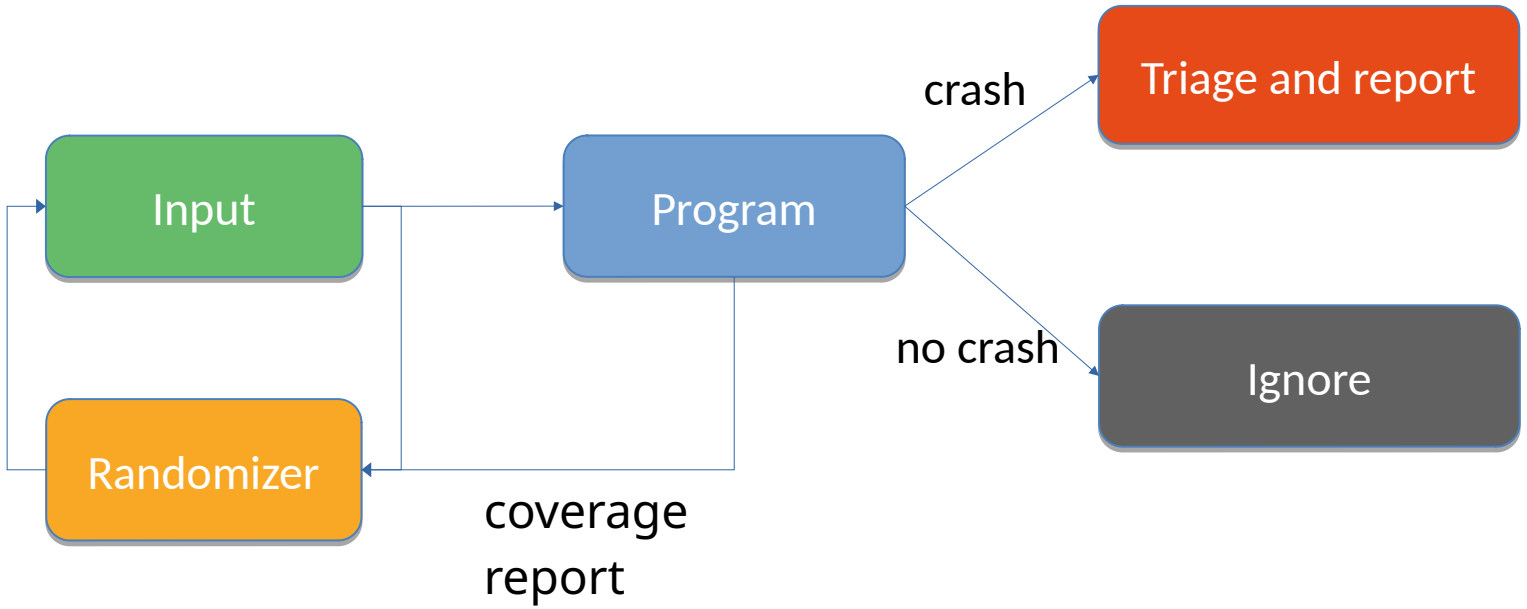
- Designed to be baked into your target application
- Part of LLVM, easily used when building a target with clang
- Coverage guided
- Requires source
- Extremely fast as it's in-memory fuzzing
  - Not dropping files to disk every iteration
- Similar to AFL it's corpus based
  - Need to have some well formed inputs to start with

ARCHITECTURAL IMPROVEMENTS TO FUZZING

COVERAGE GUIDED FUZZING

- Gather which code has been hit based on an input
- Input saved when a new unique codepath is observed
- Input is used as a basis for future inputs
- One of the biggest improvements that can be made to a fuzzer
- Can ultimately turn exponential complexity into linear complexity

COVERAGE GUIDED FUZZING



In coverage guided fuzzing, the randomizer knows whether an input has produced a new path in the code. If this happens, then the new input is saved in the corpus that is mutated by the randomizer, otherwise it is discarded.

CRASH AMPLIFICATION

Problem 1: sometimes the program just won’t crash

Solution: Increase program sensitivity to malformed input

- ASAN / PageHeap / Electric Fence
- Use heatmaps to direct the fuzzing procedure to generate more crashes

Problem 2: sometimes the bugs we want to find are not crashing bugs, but logical errors

Solution: Add hooks to find logic bugs (e.g. crash on auth success)

Limitations:

- Many programs won't start with ASAN
- Some incorrect memory access does not result in crashes



ROLL YOUR OWN FUZZER

LET’S ROLL OUR OWN FUZZER

- We target a faulty implementation of EXIF (an image metadata reader) https://github.com/hoheinzollern/exif-fuzz
- We need to define:
  - A procedure to read and mutate a set of inputs
  - A procedure to call the target program and check for crashes
  - A procedure to save crashes

SKELETON

```
def get_bytes(filename):
    f = open(filename, "rb").read()
    return bytearray(f)

def create_new(data):
    f = open("mutated.jpg", "wb+")
    f.write(data)
    f.close()

N = 100000

def exif(counter,data):
    command = "./exif mutated.jpg -verbose"
    out, returncode = run(command, withexitstatus=1)
    if b"ERROR" in out:
        f = open("crashes/crash.{}.jpg".format(str(counter)), "ab+")
        f.write(data)
        f.close()
    if counter % 100 == 0:
        print(counter, end="\r")

def fuzz(filename):
    for counter in range(N):
        data = get_bytes(filename)
        mutated = mutate(data)
        create_new(mutated)
        exif(counter,mutated)
```

Let’s read and write a test file

… and call the executable (if the program errors out, we save the input as crashing)

Fuzzing “just” amounts to mutating the original file N times and checking for failures

BIT-FLIPPIN' MUTATION

```
def bit_flip(data):
    num_of_flips = int((len(data) - 4) * .01)
    indexes = range(4, (len(data) - 4))
    chosen_indexes = random.sample(indexes, num_of_flips)
    for x in chosen_indexes:
        data[x] ^= 1 << random.randint(0,7)
    return data
```

This code mutates the original file by flipping 1% of the original bits

Next, we see how to change some bytes in more meaningful ways...

GYNVAEL'S MAGIC NUMBERS

- Gynvael Coldwind 'Basics of fuzzing' enumerates several 'magic numbers' that typically produce errors
- These numbers relate to data type sizes and arithmetic-induced errors
  - `0xFF 0x7F 0x00 0xFFFF 0x0000 0xFFFFFFFF 0x00000000 0x80000000` (minimum 32-bit int) `0x40000000` (just half of that amount) `0x7FFFFFFF` (max 32-bit int)
- If used as parameters to `malloc()` or other array operations, overflows are common
- For instance `0x1` plus `0xFF` on a one-byte register overflows to `0x00` and can produce unintended behavior
- HEVD actually has an integer overflow bug similar to this concept.
- If we choose to use `0x7FFFFFFF` as the magic number then we need to change four bytes

MAGIC NUMBER MUTATION

```
def magic(data):

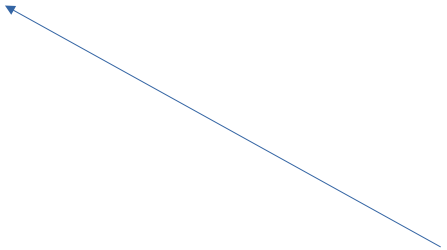
    magic_vals = [(1, 255), (1, 255), (1, 127), (1, 0), (2, 255), (2, 0),
                  (4, 255), (4, 0), (4, 128), (4, 64), (4, 127) ]

    (picked_size, picked_magic) = random.choice(magic_vals)

    picked_index = random.randint(4, len(data)-4)

    for i in range(picked_size):
        data[picked_index + i] = picked_magic

    return data
```



This function replaces random bytes with Gynvael's magic numbers

MUTATION - PUTTING IT TOGETHER

```
def mutate(data):
    f = random.choice([bit_flip, magic])
    return f(data)
```

Finally! the mutation procedure just randomly does some bit flipping or inserts Gynvael's magic numbers

TRIAGING

This is the last bit needed for a full fuzzer:
We record whether the program crashed (with a heap buffer overflow or a segmentation fault)

```
def get_files():
    return os.listdir("crashes/")

def triage_files(files):
    for x in files:
        original_output = os.popen(f"./exif crashes/{x} -verbose
2>&1").read()
        output = original_output

        # Getting crash reason
        crash = "SEGV" if "SEGV" in output else "HBO" if "heap-
buffer-overflow" in output else None

        if crash == "HBO":
            output = output.split("\n")
            counter = 0
            while counter < len(output):
                if output[counter] == "====":
                    target_line = output[counter + 1]
                    target_line2 = output[counter + 2]
                    counter += 1
                else:
                    counter += 1
            target_line = target_line.split(" ")
            address = target_line[5].replace("0x","")


            target_line2 = target_line2.split(" ")
            operation = target_line2[0]
```

```
        elif crash == "SEGV":
            output = output.split("\n")
            counter = 0
            while counter < len(output):
                if output[counter] == "==============":
                    target_line = output[counter + 1]
                    target_line2 = output[counter + 2]
                    counter += 1
                else:
                    counter += 1
            if "unknown address" in target_line:
                address = "00000000"
            else:
                address = None

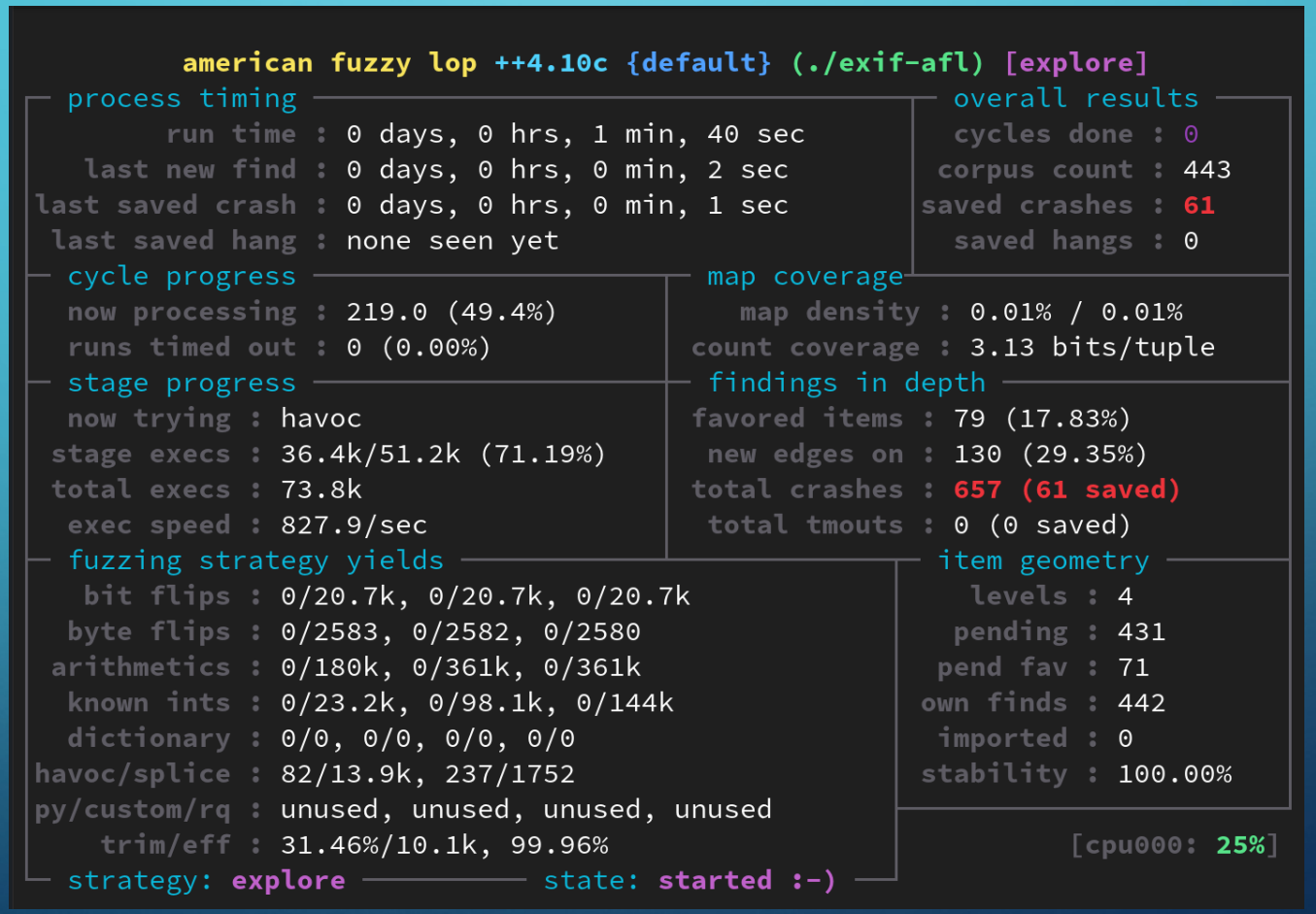
            if "READ" in target_line2:
                operation = "READ"
            elif "WRITE" in target_line2:
                operation = "WRITE"
            else:
                operation = None


        if crash:
            log_name = (x.replace(".jpg","")
    + "." + crash + "." + address + "." + operation)
            f = open(log_name,"w+")
            f.write(original_output)
            f.close()
```

RUN THE FUZZER

- Get the full fuzzer here:
  https://gist.github.com/hoheinzollern/7550ddd9a02178f96511ac68eeaa4f03
- Download the target:
  ```
  $ git clone https://github.com/hoheinzollern/exif-fuzz
  ```
- Compile it:
  ```
  $ cd exif-fuzz & make
  ```
- Run the fuzzer:
  ```
  $ python caveman.py
  ```

Fuzzing with AFL++



AFL++

- Now, let’s try to fuzz the same program with AFL
- Download AFL++:

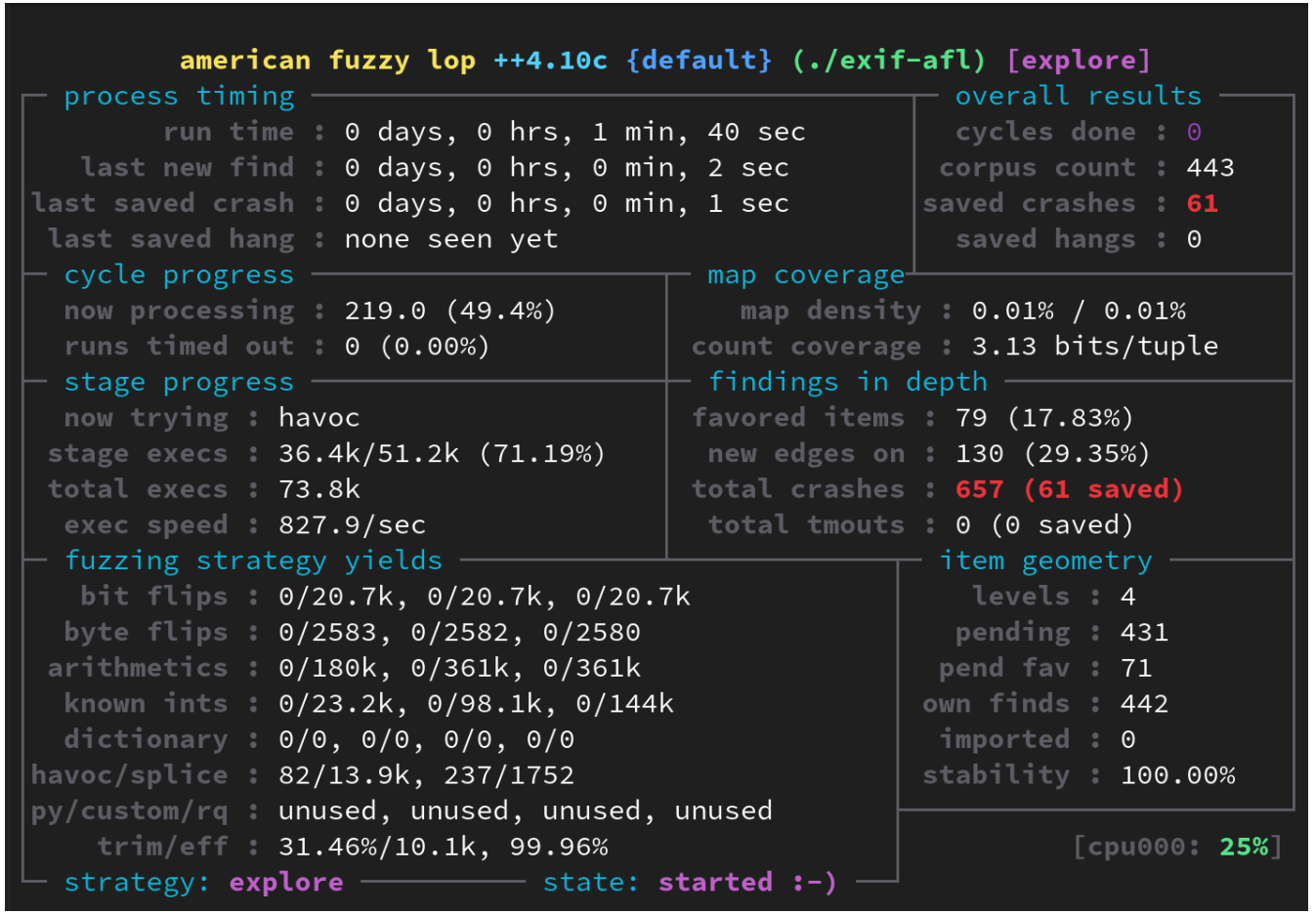
  https://github.com/AFLplusplus/AFLplusplus
- From the exif repository, checkout the afl branch and recompile:

```
$ git checkout afl && make clean && make
```

- Run the program with AFL++:

```
$ afl-fuzz -D -i in -o out ./exif @@
```

AFL++ RUNNING


american fuzzy lop ++4.10c {default} (./exif-afl) [explore]
process timing
run time : 0 days, 0 hrs, 1 min, 40 sec
last new find : 0 days, 0 hrs, 0 min, 2 sec
last saved crash : 0 days, 0 hrs, 0 min, 1 sec
last saved hang : none seen yet
overall results
cycles done : 0
corpus count : 443
saved crashes : 61
saved hangs : 0
cycle progress
now processing : 219.0 (49.4%)
runs timed out : 0 (0.00%)
map coverage
map density : 0.01% / 0.01%
count coverage : 3.13 bits/tuple
stage progress
now trying : havoc
stage execs : 36.4k/51.2k (71.19%)
total execs : 73.8k
exec speed : 827.9/sec
findings in depth
favored items : 79 (17.83%)
new edges on : 130 (29.35%)
total crashes : 657 (61 saved)
total tmouts : 0 (0 saved)
fuzzing strategy yields
bit flips : 0/20.7k, 0/20.7k, 0/20.7k
byte flips : 0/2583, 0/2582, 0/2580
arithmetics : 0/180k, 0/361k, 0/361k
known ints : 0/23.2k, 0/98.1k, 0/144k
dictionary : 0/0, 0/0, 0/0, 0/0
havoc/splice : 82/13.9k, 237/1752
py/custom/rq : unused, unused, unused, unused
trim/eff : 31.46%/10.1k, 99.96%
item geometry
levels : 4
pending : 431
pend fav : 71
own finds : 442
imported : 0
stability : 100.00%
[cpu000: 25%]
strategy: explore
state: started :-)

EXERCISE

Triage the files in the output folder, are there crashes? Use GDB to find issues in the code that generated the crash.

NON-BINARY TARGETS? WE'VE GOT YOU COVERED!

- REST APIs are also very interesting targets for fuzzing
- Try Restler or APIFuzzer
- Disclaimer: never fuzz live environment!
- API fuzzing on live environments is bad practice and can get you in trouble

WHERE TO GO FROM HERE?

- Try the exercises on fuzzing that you can find in Haaukins, can you solve them using AFL++?
- In the next slide you can find references to further reading material

FURTHER MATERIAL ON FUZZING

Papers

- An empirical study of the reliability of UNIX utilities
- The Art, Science, and Engineering of Fuzzing
- AFL++: Combining Incremental Steps of Fuzzing Research

Practical

- Fuzzing Like a Caveman
- OWASP fuzzingguide
- Fuzzgoat, An application to testfuzzers

Tools and fuzzers

- AFL (mutation, coverage-guided)
- AFL++ (more advanced than above, a fork)
- LibFuzzer (advanced, llvm-based)
- Radamsa (mutational, test-case generator, prototyping)
- Fuzzowski (Network)
- Driller
- Restler (web API)
- APIFuzzer (web API)

SYMBOLIC EXECUTION

WHAT IS SYMBOLIC EXECUTION?

- **Symbolic execution** is a technique to *explore different paths of a program* by executing it with symbolic rather than concrete values.
- Instead of using actual inputs, the **program is executed with variables**, and taking different program paths will result in different constraints on these values
- A **costraint solver** (e.g. an SMT solver) is then used to check whether a set of constraints on the variable is satisfiable, in which case a model (a set of concrete values for the program) is found

An example program

```c
int main(void) {
  char buf[32];

  char *data = read_string();
  unsigned int magic = read_number();

  if (magic == 0x31337987) {
    memcpy(buf, data, 100);
  }

  if (magic < 100 && magic % 15 == 2 &&
      14 magic % 11 == 6) {
    memcpy(buf, data, magic);
  }

  int count = 0;
  for (int i = 0; i < 100; i++) {
    if (data[i] == 'Z') {
      count++;
    }
  }

  if (count >= 8 && count <= 16) {
    memcpy(buf, data, count*20);
  }

  return 0;
}
```

Difficult to check with fuzzing

Buffer overflow

Only solution: 17

Potential path explosion

Buffer overflow

HOW TO ANALYZE THIS PROGRAM?

We could…

- Test the program
  - Lots of manual labor, hard to make sure that we cover all the edge cases
- Fuzz the program
  - Some code paths are hard to reach by pure random guesses
- Static analysis
  - Relies on overapproximation, has often false-positives/negatives
- **Symbolic execution**
  - Let’s explore this idea!

SYMBOLIC EXECUTION

Take this example program:

```
if (x >= 0) {
  if (x < 10) return A[x];
  else return -1;
} else {
  return -2;
}
```

Reachable if x >= 0 && x < 10

Reachable if x >= 0 && x >= 10

Reachable if x < 0

Core idea: instead of running the program directly on concrete inputs, explore all possible execution paths and collect the constraints on the input values that allow to reach those paths

HOW TO SOLVE THESE CONSTRAINTS?

- Satisfiability Modulo Theory (SMT)
  - Many concrete tools available: Z3, CVC5, etc.
- Idea
  - Express the problem logically as a formula of the type
  - (a || b) && (c || d) && …
  - Where each literal a,b,c,d,… can appear positive or negative and support a specific theory
- Commonly supported theories:
  - linear arithmetic, nonlinear arithmetic, bitvectors, arrays, datatypes, quantifiers, strings

EXAMPLE PROGRAM

Let’s take our program and convert each path to Z3 constraints

```
if (x >= 0) {
  if (x < 10) return A[x];
  else return -1;
} else {
  return -2;
}
```

```
from z3 import *
x = Real('x')
s = Solver()
s.add(x >= 0)
s.add(x < 10)
s.check() → sat
s.model() → [x = 1]
```

```
from z3 import *
x = Real('x')
s = Solver()
s.add(x >= 0)
s.add(Not(x < 10))
s.check() → sat
s.model() → [x = 10]
```

```
from z3 import *
x = Real('x')
s = Solver()
s.add(Not(x >= 0))
s.check() → sat
s.model() → [x = -1]
```

HOW TO DEAL WITH LOOPS?

- Consider the following program:

```
int count = 0;
for (int i = 0; i < 100; i++) {
  if (data[i] == 'Z') {
    count++;
  }
}
```

- The program suffers from ***path explosion***: each iteration of the for loop can enter or not the if branch, leading to 2^{100} possible executions
- To solve this we can limit the number of paths :
  - **Loop bounding** is one technique where loops are executed up to a bounded number of iterations
  - **Heuristic-based exploration**: various heuristics can be used to prioritize the search (e.g. the default one is DFS, always choosing the true branch first).



SYMBOLIC EXECUTION WITH ANGR.IO

WHAT IS ANGR?



- Angr is a binary analysis framework that provides a suite of tools and libraries for analyzing and manipulating binary programs that is capable of doing symbolic execution
- We will use angr to try out symbolic execution on some real programs

INSTALLING ANGR:

- To obtain angr, run these commands on a Linux shell:

```
$ virtualenv env
$ source env/bin/activate
$ pip install angr
```

- Clone the angr-examples repository to get the examples:

```
$ git clone git@github.com:angr/angr-examples.git
$ cd angr-examples
```

FAUXWARE

- Let's try a simple example, fauxware
- We have the following authentication function:

```
int authenticate(char *username, char *password)
{
        char stored_pw[9];
        stored_pw[8] = 0;
        int pwfile;
        if (strcmp(password, sneaky) == 0) return 1;
        pwfile = open(username, O_RDONLY);
        read(pwfile, stored_pw, 8);
        if (strcmp(password, stored_pw) == 0) return 1;
        return 0;
}
```

Here's a backdoor!

SOLVE SCRIPT

- Let’s try to find the backdoor!
  - let’s create a project, loading the binary
    ```
    p = angr.Project('fauxware', auto_load_libs=False)
    ```
  - add an entry state and setup the simulation
    ```
    state = p.factory.entry_state()
    sm = p.factory.simulation_manager(state)
    ```
  - we run symbolic execution until we reach a branch where both sides are satisfied *(this is the backdoor check)*
    ```
    sm.run(until=lambda sm_: len(sm_.active) > 1)
    ```
  - finally, let’s check the possible solutions
    ```
    input_0 = sm.active[0].posix.dumps(0)
    input_1 = sm.active[1].posix.dumps(0)
    print(input_0, input_1)
    ```

CTF GAME: WYVERN

- Let’s check csaw_wyvern, which was an exercise given at the CSAW CTF 2015
- Open the binary wyvern with a reverse engineering tool and check the main function
- Let’s look at the call graph

TESTING THE BINARY

- Let’s try to run the binary. We get the following output:

```
+-------------------------+
|    Welcome Hero         |
+-------------------------+

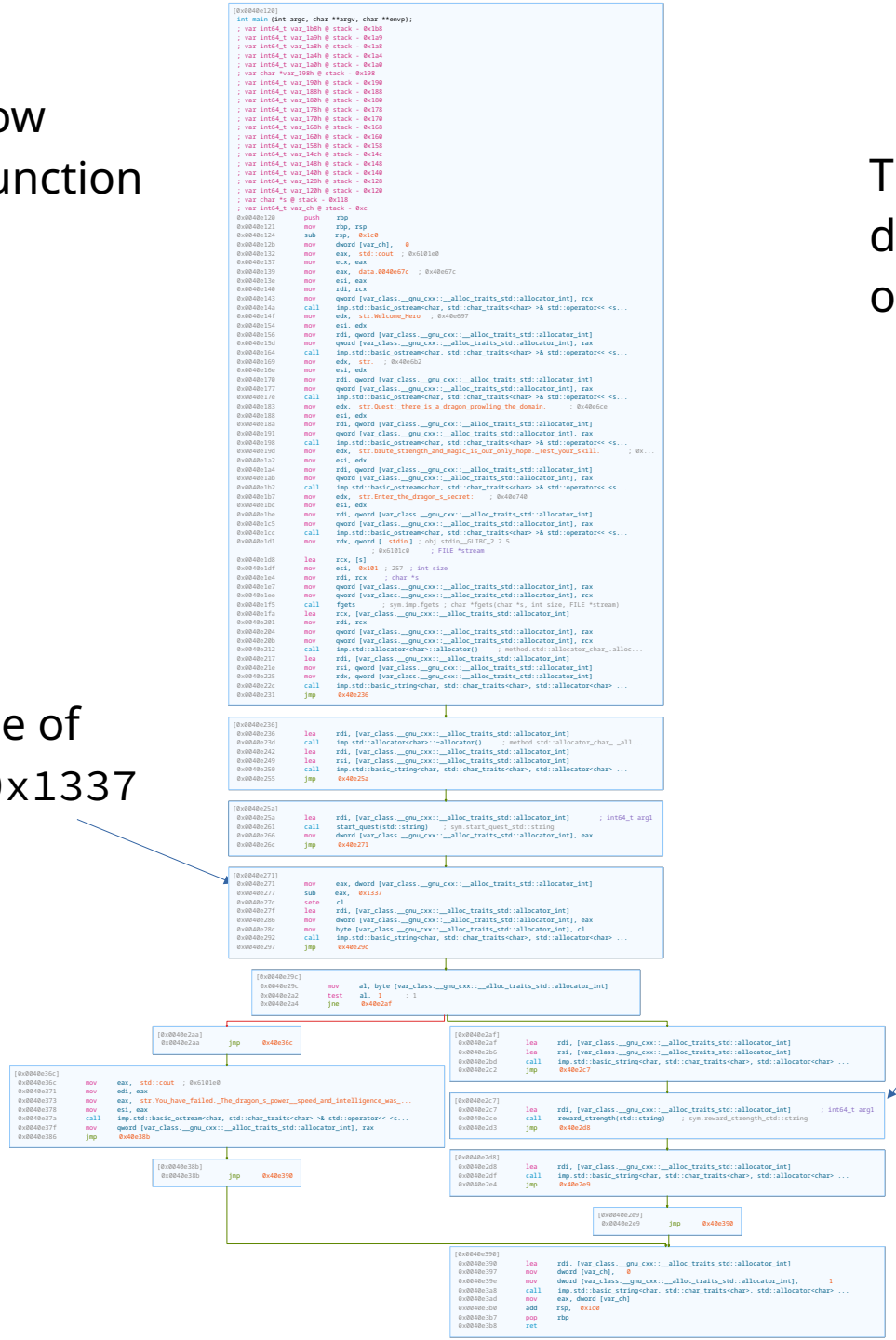
[!] Quest: there is a dragon prowling the domain.
brute strength and magic is our only hope. Test your
skill.

Enter the dragon's secret: secret!

[-] You have failed. The dragon's power, speed and
intelligence was greater.
```

- Let’s try to find the secret with angr

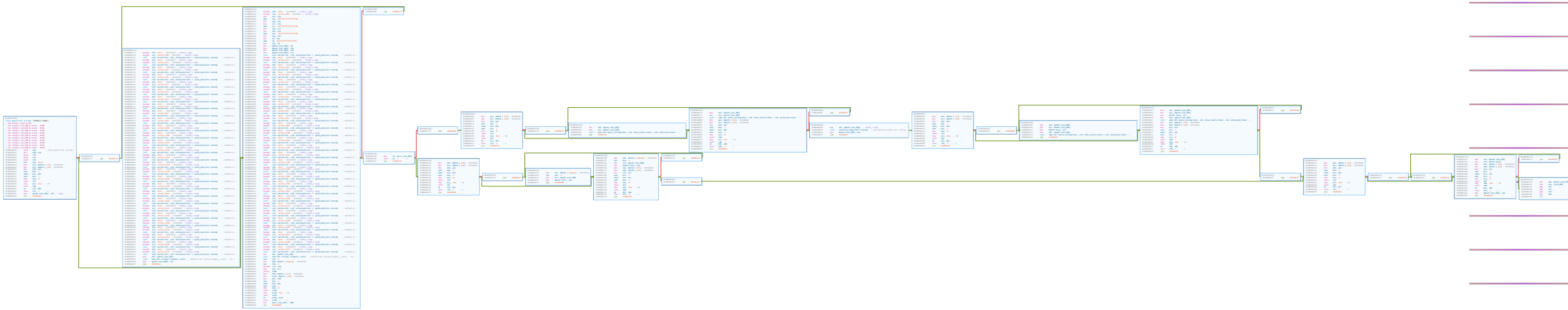
Here is the control flow graph for the `main` function



This is a big graph and may look daunting, but in fact there's only one branch at the end.

To get there, the return value of `start_quest` needs to be `0x1337`

We want to reach this branch



Ok, this looks complex…
Let’s see if we can find the solution with angr

SOLVE IT WITH ANGR

- We make a project, as usual

```
p = angr.Project('wyvern')
```

- Let's add a bit vector as input, the length of expected input is 28 bytes long

```
flag_chars = [claripy.BVS('flag_%d' % i, 8) for i in range(28)]
flag = claripy.Concat(*flag_chars + [claripy.BVV(b'\n')])
```

- Let's initialize the state. Since it's a C++ program, we need a special setup. stdin=flag sets the input to the program as the symbolic bitvector that we just created.

```
st = p.factory.full_init_state(
        args=['./wyvern'],
        add_options=angr.options.unicorn,
        stdin=flag,
)
```

- Constrain the first 28 bytes to be non-null and non-newline:

```
for k in flag_chars:
    st.solver.add(k != 0)
    st.solver.add(k != 10)
```

- We run the simulation

```
sm = p.factory.simulation_manager(st)
sm.run()
```

- Finally, we get the stdout of every path that reached an exit syscall. The flag should be in one of these!

```
out = b''
for pp in sm.deadended:
    out = pp.posix.dumps(1)
    if b'flag{' in out:
        return next(filter(lambda s: b'flag{' in s, out.split()))
```

WHERE TO GO FROM HERE

- Try the exercises on symbolic execution offered in Haaukins, can you find the attacks using angr?
- In the next slide you can find pointers to further material.

REFERENCES

- Angr documentation:
  https://docs.angr.io/en/latest/

Other symbolic execution tools:

- The binary analysis platform
  https://github.com/BinaryAnalysisPlatform/bap
- Java PathFinder
  https://github.com/javapathfinder/
- And more:
  https://github.com/ksluckow/awesome-symbolic-execution

Further reading

- A survey of symbolic execution techniques
  https://arxiv.org/pdf/1610.00502.pdf
- (State of) The Art of War: Offensive Techniques in Binary Analysis
  https://sites.cs.ucsb.edu/~vigna/publications/2016_SP_angrSoK.pdf

FIND OUT MORE

https://gamess.dk/

WHO IS BEHIND GameSS

Partners behind the project

SDU

DTU

IT UNIVERSITY OF CPH

AALBORG UNIVERSITET

Happy 42

Collaborators

ERHVERVSAKADEMI AARHUS

kea
KØBENHAVNS ERHVERVSAKADEMI

Dania
ERHVERVSAKADEMI

UCN

UCL Erhvervsakademi og Professionshøjskole

Supported by

Uddannelses- og Forskningsministeriet

Ministry of Higher Education and Science Denmark