

# Fuzz Testing

Lars Bo Frydenskov

Department of Computer Science  
Aalborg University

29 MAR 2023



**AALBORG UNIVERSITY**  
DENMARK

# Learning Goals

After today's lecture you should be able to

- ... **explain** and **discuss** why and how fuzz testing is used to find unexpected behaviour in software
- ... **explain** and **discuss** the different fuzz testing methods, including
  - blackbox
  - fuzzing using instrumentation
  - mutation-based and generation-based
  - assertion-based
  - whitebox
- ... **apply** fuzz testing via AFL++ on simple CLI programs

# Easy to test?

## Example 0

```
uint8_t tab[0x1ff + 1];

uint8_t target(int32_t value){
    if(value < 0) {
        return 0;
    }

    int32_t i = value * 0x1ff / 0xffff;
    if(i >= 0 && sizeof(tab) > i){
        return tab[i];
    }
    return 0;
}
```

# Easy to test?

## Example 0a

```
uint8_t target(int32_t value){  
    if(value < 0) {  
        ...  
    }  
    ...  
    if(i >= 0 && sizeof(tab) > i){  
        ...  
    }  
    ...  
}
```

# Easy to test?

## Example 0

```
uint8_t tab[0x1ff + 1];

uint8_t target(int32_t value){
    if(value < 0) {
        return 0;
    }

    int32_t i = value * 0x1ff / 0xffff;
    if(i >= 0 && sizeof(tab) > i){
        return tab[i];
    }
    return 0;
}
```

- Integer overflow and optimisation removes check!

Original Example: [blog.pkh.me](https://blog.pkh.me)

# Why Fuzz Testing?

Secure, Secure, Secure(?)

## Critical Systems: it goes without saying

- Military
- Healthcare
- Financial Systems
- Infrastructure

## What about less critical systems?

- Computer/mobile applications
- IoT and embedded devices

## Tests, tests, tests!

System test, unit test, user test and etc.

- expensive
- time consuming

We still have insecure systems and software. More tests?

# Why Fuzz Testing?

B. Miller and Automated Test

- Barton Miller introduced the keyword "**fuzz**" in 1988
- An attempt to make automated test for UNIX command line utilities
- Doing so by generating random input and observing crashes



## Fuzz testing works!

Miller et al. crashed up to 33% of utilities tested back in 1988. In 2020 the same method was applied on UNIX-bases systems and there was found upto 19% failure rates

# Why Fuzz Testing?

Fuzzing today - Red and Blue

Evolved since 1988

Fuzz testing have been widely adopted and applied. IT giants such as Google and Microsoft are using it as part of the CI/CD

- Widely adopted in the red team community
- Recently increasing popularity tool for blue teamers



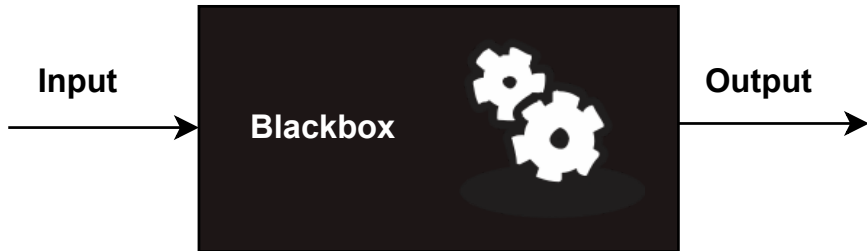
- Today, fuzz testing is better known as **fuzzing**



# Blackbox Fuzzing

# Blackbox Fuzzing

## Input and Output



We, the fuzzer, can only observe input and output and is **randomly** guessing inputs!

- How long does random take?

# Blackbox Fuzzing

## Banality of Random

### Example 1

```
int fuzzing_target(int input){  
    int output = 0;  
  
    if (input == 2023){  
        abort();  
    }  
  
    return output;  
}
```

- What is the flaw?

# Blackbox Fuzzing

## Banality of Random

### Example 1

```
int fuzzing_target(int input){
    int output = 0;

    if (input == 2023){
        abort();
    }

    return output;
}
```

- What is the flaw?

A integer value in C consist of four bytes, which can represent  $2^{32} = 4,294,967,295$  different values. The probability of random guessing 2023 is then 1 : 4,294,967,295

# Blackbox Fuzzing

## Banality of Random

### Example 2

```
int fuzzing_target(int input_a, int input_b, int input_c){
    int output = 0;

    if (input_a == 2023){
        if (input_b == 03){
            if (input_c == 29){
                abort();
            }
        }
    }
    return output;
}
```

Well, the probability of random generating the correct sequence of integer values is  $1 : 2^{96}$  – which is small

# Blackbox Fuzzing

Still Random but Less

What is most likely to provoke an error?

- "A" or "AA"
- "SejeReje" or ">!S3JE0?r@Je#{';"
- "0" or 0x00
- ":-)" or "😊"

Random inputs can still be evaluated and prioritised - to improve accuracy

# Blackbox Fuzzing

## Pros and Cons

### Pros:

- It works!
- Easy to implement – or reuse

### Cons:

- Guarantee nothing
- Might never reach interesting code

## Application of Blackbox

Blackbox fuzzing excels when the target is unknown:

- Pentesting and Red Teaming
- APIs, Embedded systems etc.
- For everybody :)

# Test Cases



# Test Cases

## Example

Test Cases – inputs, seeds

### Example 3

```
int fuzzing_target(char* input){
    int output = 0;
    if (input[0] == 'F'){
        output++;
        if (input[1] == 'U'){
            output++;
            if (input[2] == 'Z'){
                output++;
                if (input[3] == 'Z'){
                    output++;
                    exciting_stuff(input);
                }
            }
        }
    }
    return output;
}
```

# Test Cases

## Mutation-based Fuzzing

Mutation of an initial test case:

FUZZing is great

Is done by applying a mutation operations, such as:

- prefixing and suffixing interesting values
- Removing sub strings
- bit-flipping
- adding or subtracting

Resulting in:

fuzzING IS GREAT

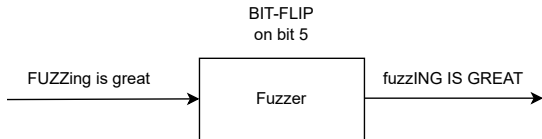
FZinis gat

FUZZing is greatFUZZing is great

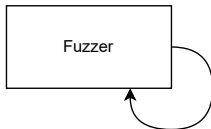
...

# Test Cases

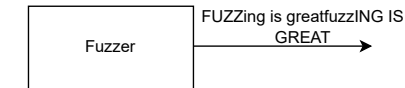
## Mutation-based Fuzzing



Reusing test case as  
interesting value



fuzzING IS GREAT



Suffixing with interesting value

# Test Cases

## Mutation-based Fuzzing

Using output as an indicator of interesting test cases

- xxxxxxxx : 0
- Fxxxxxxx : 1
- FUxxxxxx : 2
- FUZxxxxx : 3
- FUZZxxx : 4+y

### Example 3

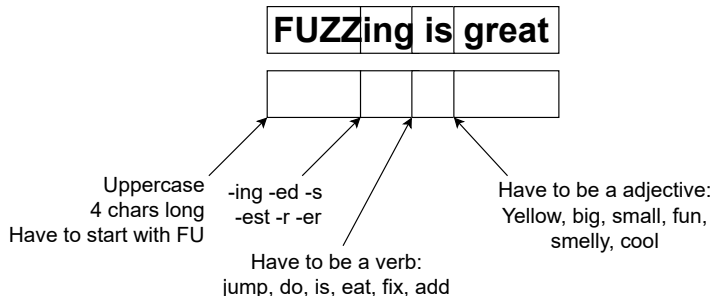
```
int fuzzing_target(char* input){
    int output = 0;
    if (input[0] == 'F'){
        output++;
        if (input[1] == 'U'){
            output++;
            if (input[2] == 'Z'){
                output++;
                if (input[3] == 'Z'){
                    output++;
                    exciting_stuff(input);
                }
            }
        }
    }
    return output;
}
```

# Test Cases

## Generation-based Fuzzing

### Generation-based Fuzzing - Aware of input structure

Some rules might apply to the input structure, these are used in generation-based fuzzing



- Generation of test cases that satisfy the rule set

# Test Cases

## Generation-based vs. Mutation-based

### Generation-based

- Be aware of input structure
- Better accuracy
- Less Random
- Biased?

### Mutation-based

- Only needs initial test case
- Reusable
- More random

### Fuzzing harness

Fuzzing harness is the code which is needed in order to start fuzz testing, this can vary a lot depending on which methods and tools used

## Time to make your first blackbox fuzz test with AFL++!

Take a look on the following exercises:

- `exercise/1/a`
- `exercise/1/b`
- `exercise/1/c`
- `exercise/1/d`

The exercises can be found here: [github.com/larsbpf/fuzzing-lecture](https://github.com/larsbpf/fuzzing-lecture)

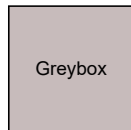
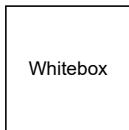
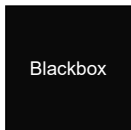
- Remember the changes you make in the Docker Container is not **saved**
- Rerunning the buildscript **removes** the existing container and makes a new with the current contents of the exercise folder

# Greybox Fuzzing



# Greybox Fuzzing

Grey is the New Black



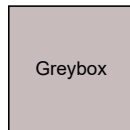
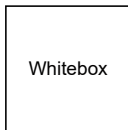
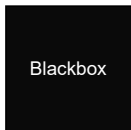
## Box, Box, Box

The three boxes indicate how much the tester knows about the target.  
The tester has access to...

- Black: only input and output
- White: source code (everything)
- Grey: something in between?

# Greybox Fuzzing

Grey is the New Black



## Box, Box, Box

The three boxes indicate how much the tester knows about the target.  
The tester has access to...

- Black: only input and output
- White: source code (everything)
- Grey: something in between? – it's a bit fuzzy

## 50 shades of Greybox

Greybox is very loosely defined which also result in a very broad interpretation of the term

# Not-Quite-Greybox Fuzzing

American Fuzzy Lop

American Fuzzy Lop better known as AFL

AFL is not self-proclaimed greybox, but is sometimes referred to as one. It have been forked 500+ times and parent most community acknowledge fuzzers

- Low effort
- Highly optimised

Directed & Coverage-based

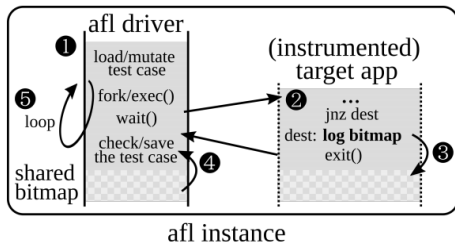
Directed fuzzers are trying to discover some specific block in a program, where as coverage-based try to discover as much as possible

# Not-Quite-Greybox Fuzzing

## American Fuzzy Lop - Instrumentation and Mutation

### Repeating

- (1) Reading and mutating inputs
- (2) Launching the target application
- (3) Executing and recording runtime coverage
- (4) Bookkeeping results



### Instrumentation by compiler

AFL uses a custom compiler in order to insert logging calls in the targeted application

# American Fuzzing Lop

## Pros & Cons

### Pros:

- Highly optimised
- Easy-to-use (using mutation-based)
- Offers blackbox fuzz testing and fuzz testing with instrumentation
- Random

### Cons:

- Hard to master
- Need to instrument source code in order to work optimal
- Limited to C and C++
- Random

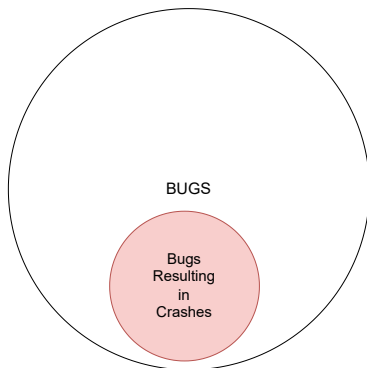
# Assertion and Exposing

# Searching For More

## Assertion-Based Fuzzing

### To Crash or not to Crash?

'Classic' fuzz testing methods only search for memory related bugs. Bugs that results in crashes.



- What is an example of a bug that does not result in a crash?

# Searching For More

## Assertion-Based Fuzzing

- What kind of behaviour does we expect?

### Example 4

```
struct UserProfile {
    unsigned int id;
    char firstname[32];
    char lastname[32];
    int balance;
}

struct User* withdraw(struct UserProfile* user){
    /* Code handling the withdrawal process */
    return user;
}
```

### Assertions - Make sure ... or die!

An assertion is used to check and expression and if it evaluates false then crash otherwise continue



# Searching For More

## Assertion-Based Fuzzing

### Example 4a

```
struct User* withdraw(struct UserProfile* user){  
    /* Code handling the withdraw process */  
  
    assert(new_balance < prev_balance);  
    assert(user->balance >= 0);  
    assert(user->id == id);  
    /* etc. */  
  
    return user;  
}
```

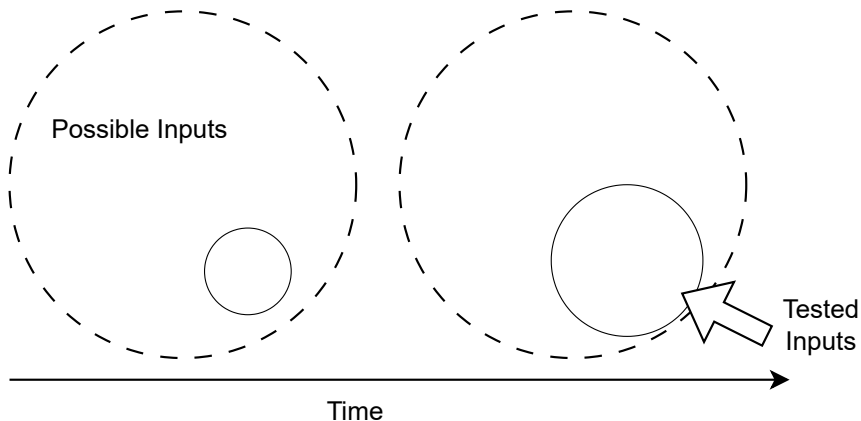
### Assertion-based Fuzz Testing

Assertions make it possible to fuzz test logic, and makes fuzz testing to a much stronger tool

# Ignore the Irrelevant

Faster is More

- How big can an input be?
- How many different inputs exists?
- How long does random take?



# Ignore the Irrelevant

## Removing IO and Sleeps

### Example 5

```
int main(){
    char* input[64];
    int sleep_time;

    printf("Welcome to the sleeping machine\n");
    printf("How long you want to sleep?\n:");
    scanf("%d",sleep_time);

    sleep(sleep_time);

    printf("Oooh, no. A program that needs to be fuzzed!\n"
           "Give an input:");
    scanf("%63s", input);

    fuzzing_target(input);

    return 0;
}
```

# Ignore the Irrelevant

## Removing IO and Sleeps

### Example 5a

```
int main(){
    char* input[64];
    int sleep_time;

    printf("Welcome to the sleeping machine\n");
    printf("How long you want to sleep?\n:");
    scanf("%d",sleep_time);

    // sleep(sleep_time);

    printf("Oooh, no. A program that needs to be fuzzed!\n"
           "Give an input:");
    scanf("%63s", input);

    fuzzing_target(input);

    return 0;
}
```

# Ignore the Irrelevant

## Removing IO and Sleeps

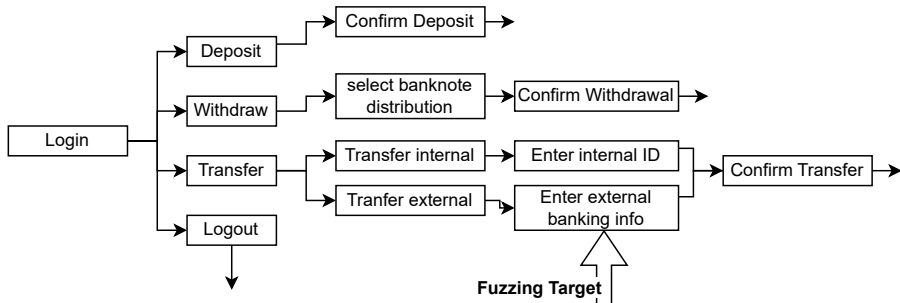
### Example 5b

```
int main(){
    char* input[64];
    scanf("%63s", input);
    fuzzing_target(input);

    return 0;
}
```

# Ignore the Irrelevant

## Exposing Functions



## Example 6

```
int main(){
    char* input[1024]; /* Initialising input buffer */
    fgets(input, sizeof(input), stdin); /* Reading from terminal */
    check_ex_banking_info(input); /* Calling fuzzing target */
    return 0;
}
```

## AFL++ a coverage-based with the help of instrumentation

Take a look on the following exercises:

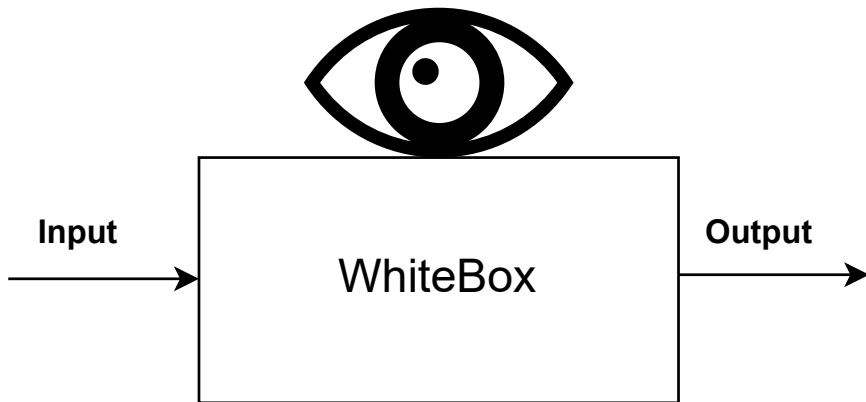
- `exercise/2/a`
- `exercise/2/c`
- `exercise/4/a`

# Whitebox Fuzzing



# Whitebox Fuzzing

## Fuzzing and Beyond



We know and are aware of everything including the source code and specification

# Whitebox Fuzzing

## Fuzzing and Beyond

### The Curse of Knowledge

Analyses such as symbolic executions is complex and often suffers from complexity problems. Here is the main problems of symbolic execution:

- Path explosion - the number of paths rise exponential with control structures
- Determining if a formula can be satisfied - SAT or harder

### Ignorance is Bliss

Whitebox fuzzers uses symbolic execution to find interesting paths with combination of the *traditional* fuzz technique of using random inputs, to achieve greater depth on less time

# Whitebox Fuzzing

## Fuzzing and Beyond

### Pros:

- Very precise
- Can cover much larger code bases
- Able to find logic related errors

### Cons:

- Time and memory consuming
- For experts!

### To be Continued...

Whitebox fuzzing is still evolving, since it combines two non obvious ideas: static/dynamic analyses of a application and testing with random generation of inputs.