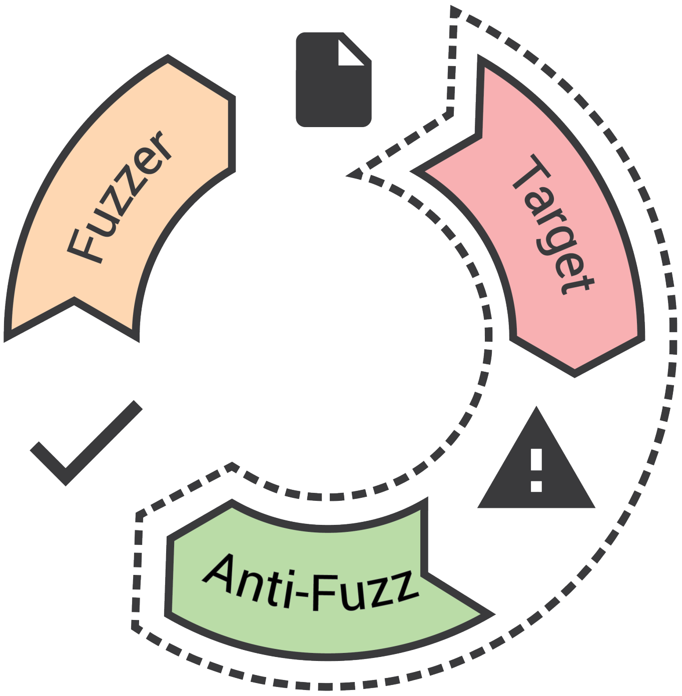
CHALMERS
UNIVERSITY OF TECHNOLOGY



# Escaping the Fuzz

Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing

Master's thesis in Computer Systems and Networks

DAVID GÖRANSSON
EMIL EDHOLM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016

# Escaping the Fuzz

Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing

DAVID GÖRANSSON
EMIL EDHOLM

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Technology
Chalmers University of Technology
Gothenburg, Sweden 2016

Escaping the Fuzz
Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing
DAVID GÖRANSSON
EMIL EDHOLM

© 2016 DAVID GÖRANSSON, EMIL EDHOLM.

| | |
|---|---|
| Academic supervisor: | Alejandro Russo |
| | Department of Computer Science and Engineering |
| | Daniel Kvist, Kasper Karlsson |
| | Omegapoint AB |
| Examiner: | Patrik Jansson |
| | Department of Computer Science and Engineering |

Master's Thesis 2016 Department of Computer Science and Engineering
Division of Software Technology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A circular flowchart showing the high level concept of anti-fuzzing. A program will, from the fuzzer's point of view, always be in a non-crashing state.

Typeset in LaTeX
Printed By TeknologTryck
Gothenburg, Sweden 2016

Escaping the Fuzz
Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing
DAVID GÖRANSSON
EMIL EDHOLM
Department of Computer Science and Engineering
Chalmers University of Technology

## Abstract

Fuzzing is used to find vulnerabilities in applications by sending garbled data as input and then monitoring the application for crashes. Over the years, this simple technique have evolved to an advanced testing technique that have been used to find a serious vulnerabilities in a wide range of applications. This thesis sets out to evaluate two state-of-the-art fuzzers and pinpoint their weaknesses. The thesis also investigates anti-fuzzing: a technique that masks crashes from fuzzers. By not detecting crashes, fuzzers become useless when it comes to detecting vulnerabilities in software. The fuzzers are tested against a test suite of security vulnerability challenges from the DARPA Cyber Grand Challenge and then against the same test suite when anti-fuzzing capabilities have been built-in. Our results show that it is relatively easy to implement and apply anti-fuzzing techniques that are able to completely mask crashes and by extension, vulnerabilities from fuzzers.

Keywords: security, fuzzing, fuzzer, anti-fuzzing, fuzz-testing

# Acknowledgements

We would like to thank our supervisor Alejandro Russo for his continuous support during this thesis project. His ideas and feedback have been an immense help in order for us to successfully complete this project. We would also like to thank our industry supervisors Daniel Kvist and Kasper Karlsson for their help and encouragement during this project. As a final thank you, we would like to thank Omegapoint AB for lending us servers and hardware that we could use in our evaluations.

David Göransson & Emil Edholm, Gothenburg, June 2016

# Contents

# 1 Introduction

The concept of a *fuzzer* was invented in the late eighties by Barton Miller as a way to do automatic testing of common Unix utilities [1, 2]. As he described the term: "I wanted a name that would evoke the feeling of random, unstructured data. After trying out several ideas, I settled on the term fuzz.". Nowadays, *fuzzing* refers to the systematic approach of sending garbled input to a target application in order to record any anomalous behaviour. A conceptual view of the fuzzing process can be seen in Figure 1.1. A fuzzer will generate data that is consumed by a target application.

Since its introduction, fuzz testing has grown to be an important tool for discovering security related bugs in software. There exists fuzzers designed to target specific protocols, such as FTP, and other more general purpose fuzzers that are not bound to a program or protocol. Many large companies have written their own fuzzers and regularly test their own software with these, including Microsoft and Google [3, 4]. However, fuzz testing is not limited to large companies. Since the fuzzing process can be run for an unlimited time, having access to a large number of CPU-cores will speed up the process of finding bugs, but that does not mean that you need a super computer to fuzz a target application or protocol.

One might imagine a small company that releases a new application, but does not have access to a large amount of servers for running the fuzz testing on. This company could level the playing field by introducing a technique in their application that makes it harder for fuzzers to target it. They can then fuzz their own software without the technique applied. This obviously has financial advantages since it

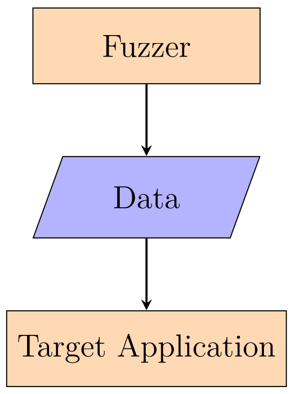


**Figure 1.1:** Conceptual overview of fuzzing process

allows the company to release their product earlier, knowing that any adversaries will have a hard time finding vulnerabilities through the use of fuzzing. Additionally, such a technique could reduce the risk of bad publicity due to potential security related issues in the released software. We call that technique anti-fuzzing.

Anti-fuzzing is a concept that is similar to obfuscation. The target binary is hardened against fuzzing, but like obfuscation it is only meant to slow down an adversary rather than stopping them. The NCC Group has released a white paper that discusses anti-fuzzing in broad terms and talks about any potential impact it may have [5], however, we have found no real research on the area.

This thesis consists of two majors parts: an evaluation of fuzzers and an evaluation of anti-fuzzing. To achieve this, the thesis will first, in Chapter 2, cover the basics of fuzzers and the fuzzing process. Thereafter, in Chapter 3, the evaluation and comparison of two state-of-the-art fuzzers are made. In the Chapters 4 and 5 we will cover the theory of anti-fuzzing and perform an evaluation of anti-fuzzing by showing how the fuzzers fairs against targets with built in anti-fuzzing capabilities. Finally, ethical considerations and a conclusion of our work is covered in Chapters 6 and 7.

## 1.1 Purpose and Goals

The purpose of this thesis is to raise awareness about issues surrounding fuzzing and anti-fuzzing. Over the years fuzzing have become easier to perform and little knowledge is required to effectively fuzz a target. However, if the user running the fuzzer is unaware of issues surrounding anti-fuzzing, he or she may take the results at face-value which may lead to a false sense of security.

The project will consist of two parts: the first part aim to test fuzzing on selected targets in a normal setup. The second part will consist of developing and evaluating the anti-fuzzing techniques. The anti-fuzzing will be done by applying the technique on the selected targets and redoing the fuzz test in the same way as in part one. The results from part one and two can be presented and compared, with respect to the developed anti-fuzzing techniques.

## 1.2 Scope and Limitations

In order to make our research relevant and not too general, we have introduced a few limitations on the scope of the thesis. These limitations mainly focus on the fuzzers and applications selected.

To keep the research worthwhile and interesting, we have chosen to target current state-of-the-art fuzzers. We also required them to be under *active development*, defined as having had an update in the last two years. For the study, we have chosen to only target publicly available fuzzers. Any non-free commercial fuzzers will be out of scope. The reasoning behind this is threefold: (1) The source code of the fuzzer is often not publicly available, therefore it is hard to determine and discuss how they work in an unbiased manner, (2) many commercial products come with license agreements that dictate how they may be used, and what information can be released about any results. There may also be restrictions on how many fuzzing instances that can be run simultaneously, and (3) commercial fuzzers cost money.

The target applications will be limited to the *nix platform. The *nix platform offers a wide range of open source applications compared to other platforms, such as Windows and OS X. When using a fuzzer such as afl-fuzz [6], which relies on injected instrumentation to work optimally, having access to source code is essential to doing a fair comparison. Moreover, being able to read the source code allows for easier debugging and understanding of the targets.

A fuzzer generates input that a target application consumes. A consequence of this is that any target application must consume some kind of input for fuzzing to be of any use. A target application may use input in different ways: (1) the target could be a file format parser, (2) take input via standard input (stdin), or (3) communicate using an interface, such as a network socket. Network applications is out of scope for this thesis.

# 2
# Fuzzing

Evaluating the performance of fuzzers require theoretical knowledge of the fuzzing process. This chapter presents the two different categories of fuzzers, describes the use of seed values, and finally goes into detail about what a fuzzing framework is.

## 2.1 Fuzzer Categories

A *fuzzer* is an application that generates input test cases for some target application [2, 7]. This can either be done by altering normal, correct inputs or by generating completely new test cases. A *fuzzing framework* can then use these generated test cases to try to induce crashing or deadlock states in the *target application* [1, 8].

A naive approach to this process is to randomly generate data as fast as possible. However, testing this data using the target application takes more time than generating it. For example, starting several thousand processes per second will incur overhead that slows the whole fuzzing process down. This means that, for the fuzzing process to be effective, it must not generate test cases that are useless and leads to work being done unnecessarily. Such test cases could be, for example, cases where the target discards the input outright because it does not pass some validation algorithm.

*Symbolic execution* is a software testing technique that is used to trace all possible executions paths and the input required to reach them [9]. This is essentially what fuzzing tries to achieve as well, but through the use of trial and error. The reason symbolic execution does not replace fuzz testing is that it is a resource-costly technique and is currently not feasible for testing large applications, in part due to the path explosion problem [10]. *Concolic execution* is a term combined from symbolic and concrete runs, where instead of only using symbolic values, an execution is also done with concrete values in order to increase performance. Current research on fuzzing has been focused on trying to combine the strengths of symbolic/concolic execution in order to eradicate the weaknesses of fuzzers [3, 11, 12, 13, 14].

Unlike static analysis tools, a fuzzer does not need access to the source code of a target application in order to work. However, access to the source code may indirectly grant a fuzzing framework observational capabilities. By, for instance, providing a feedback loop which drives the coverage of the different fuzzed inputs. Fuzzing is normally viewed as a penetration testing tool, but can also be used for finding bugs and crashes that are not security related.

There exist two major categories of fuzzers: *generational* (sometimes called grammar-based) and *mutational* [8]. The two following sections describe these categories in more detail.

### 2.1.1 Generational

A *generational* fuzzer is a fuzzer that instead of mutating existing data, generates new data based on a template or a grammar specification. By definition a generational fuzzer is also considered a smart fuzzer. The term smart fuzzer refers to the fact that the fuzzer is context aware, and has in-depth knowledge about the target's input format and flow.

#### 2.1.1.1 Template

A generational fuzzer uses a *template* which defines the structure of the data to be generated. The goal of a template is to accurately represent the data to be consumed by the target in order for the fuzzer to generate input. In many cases, applications use standards, (such as Requests for Comments, RFCs) or custom protocols, which then can be interpreted to create templates.

A template is a very detailed depiction of all possible inputs. For example, in a protocol specification, each possible field is defined along with what type of data it may contain. The expressiveness of the template is both a strength and a weakness. Creating a template is a time consuming and precarious process making it easy to make a mistake. How accurate a template represents the specification will directly affect the results of the fuzzing. A template that does not comply with the specification is less likely to achieve a high code coverage since less data will be valid. Conversely, a high quality template allows a fuzzer to generate valid data for complex fields, such as checksums or challenge-response messages.

Even though many of the applications that are fuzzed make use of standardised protocols or file formats, there does not exist any standardisation of templates. Each generational fuzzer has a different way of implementing their templates. As a results of this, templates can not easily be shared between generational fuzzers.

### 2.1.2 Mutational

A mutational fuzzer is, as the name implies, used on existing data. It takes a *seed* value and applies different mutation strategies on it. This yields new mutated seeds that can be tested against the target application. Depending on different factors, a seed may be reused for further mutations or be discarded. Note that it is the fuzzing framework that makes these decisions and not the fuzzer itself, tough in many cases the framework is built into the fuzzer. These decisions are made in the *feedback loop* (see Section 2.3). The feedback loop allows the framework to gain more information about how the target behaved, based on a certain mutation, and can make a better decision about whether or not a mutation was successful or not.

The use of a seed value means that the fuzzer does not have to know anything about the structure or input requirements of the target. A mutational fuzzer is therefore most often classified as *dumb*. The negative side of this is that many mutations are unfruitful because the fuzzer does not know what the target expects. A successful fuzzer generates or modifies data in such a way that the target will accept the input, and not outright discard it. For example, consider the application in Listing 1: a simple program that reads some value from standard input and checks the given value against a stored magic value. A mutational fuzzer will struggle with this program since the probability of making a mutation where the input corresponds to the magic value is low.

```c
#define MAGIC_STR 0xedaeda
int main(int argc, char const* argv[]) {
    char *input = readFromStdin();
    int inputInt = atoi(input);    // Convert string to int

    if(inputInt == MAGIC_STR) {
        doWork();
    } else {
        error("Invalid magic number!");
    }
    return 0;
}
```

**Listing 1:** A simple program that will not do any work unless given a magic number as input

A guided fuzzer such as American Fuzzy Lop (AFL) will have more success with this example, since if one mutation results in the `doWork()` branch, AFL will save this test case and do the following mutations based on this test case. The probability of reaching such a mutation is the same as any other mutational fuzzer, however, any subsequent mutations will always have a way of reaching the `doWork()` branch. To combat the problem of mutational fuzzers getting "stuck", Stephens et al. [11] propose a modified version of AFL whereby concolic execution is used to drive the fuzzer to new code *compartments*. In scenarios where the fuzzer is stuck in one compartment, a concolic executor takes over and guides AFL to a new compartment, if applicable. In the example of Listing 1, the concolic executor would take over from AFL after a certain time where no new paths have been reached. This produces the magic number that then allows AFL to reach further compartments in the `doWork()` function.

#### 2.1.2.1 Strategies

A mutational fuzzer employ several different strategies when mutating seeds. Different fuzzers may employ different strategies, and may perform them in different ways (e.g. where to apply them). Little research has been performed on the mutational strategies employed by fuzzers and the techniques have mainly derived from intuitive reasoning and experiments during development [15]. For example, some of the strategies have evolved to trigger commonly made programming errors, such as off-by-one errors or integer overflow. Other strategies, such as the dictionary strategy, have been developed to overcome some of the limitations of mutational fuzzers.

Many of the strategies used by different fuzzers today are alike, and can be summarised into the following categories: bit and byte flips, arithmetic operations, interesting values, extending, trimming, and dictionary. It is important to note that depending on the implementation, the strategies may produce different results.

**Bit and byte flips** One of the most simple and basic strategies. A fuzzer using this strategy will choose a part of the seed to mutate, and flip one bit or one whole byte. A single bit flip can, for example, affect the target's branching in a considerable way. Boolean values parsed from the input can be represented by a single bit. Therefore a bit flip can change a `true` value to a `false` value.

**Arithmetic operations** This strategy involves the fuzzer reading, for example, 32 or 64 bits of the data being mutated, parsing it as a number, and then performing *arithmetic operations*, such as addition, subtraction or multiplication on it. Afterwards, the number is written back in the same place the fuzzer read it from. Binary file formats and protocols generally represent numbers in the same way as programming languages, 8-, 16-, 32- or 64-bits. Performing *arithmetic operations* on chunks of the seed may incur subtle bugs the programmer had not thought to check for.

**Interesting values** A set of *interesting values* are defined in the fuzzer and then used to replace data in the seed. The set of values generally contain extreme values, such as minimum and maximum values, and values close to zero. Values may be of different bit lengths and can be formatted in both little and big endian.

**Extending** This strategy consists of *extending* a seed with more data. By allowing seeds to grow more data will be parsed and the fuzzer will reach deeper within the application, thus achieving a greater code coverage. In some cases the target application might not able to handle the excess amount of data, making it behave in an undefined way. The extending can be made in several ways, for example randomly generating data which is added to the seed or by duplicating parts of the seed.

**Trimming** This strategy involves *trimming* and removing data from a seed. This strategy can have multiple benefits. The target application can misbehave if it expects the seed to be a certain length. Alternatively, if a seed can be shortened without affecting the behaviour of the target, then the overhead reduced, which results in an increase in performance of the fuzzer. Additionally, a smaller seed which still executes the same path in the program will cause less unnecessary mutations.

Trimming can also be used to find the smallest mutation of the seed that still causes a crash. By removing excess data there will be less data to analyse while debugging the cause of the crash.

**Dictionary** To combat the limitations of mutational fuzzing, e.g. magic values as previously shown in Listing 1, some fuzzers may implement a *dictionary* strategy. This strategy borrows some concepts from generational fuzzing in that the user can specify a dictionary of common word inputs that the target application takes as input. This could, for example, be protocol keywords such as "Set-Cookie" or "Username". The dictionary strategy could be very useful in some scenarios, since the probability of getting "Username" by the use of the other mutational strategies is low.

The fuzzer will then use the *dictionary* to add new words or change existing words. For example, one could fuzz source code with the help of the *dictionary*. If the fuzzers sees the word `public`, it could replace it with `protected` if both of those words were defined in the *dictionary*, which may not be valid source code in some circumstance. Without the *dictionary*, those words would hold no special significance to the fuzzer. In that respect, this strategy is similar to the *interesting values*-strategy. This is an effective strategy at overcoming some of the limitations of mutational fuzzing.

## 2.2 Seeds

A *seed* is user provided value that is used as a starting point for the fuzzing process. Mutational fuzzers relies more on input seeds than generational fuzzers, but their use is not limited to one or the other. A set of seeds to be used as input is called *input corpus*.

Generational fuzzers can use seeds to set up defaults that should not change. Examples of this can be username and passwords fields when fuzzing network protocols. Trying to guess credentials in addition to the normal fuzzing process is obviously unnecessary.

The results achieved by a mutational fuzzer will heavily depend on the input seed provided. The chosen seed will directly correlate with the amount of achievable code coverage. This is due to the fact that the seed may only trigger some functionality in the target. For example, consider an application that extracts metadata from different media files, such as JPEG, PNG, WAV, etc. If a PNG image file is used as a seed to this application, only the parts related to PNG files will be tested. Different PNG files with different functionality may also trigger different parts of the code. For example, one image may include transparency while others do not. This is the reason why the use of multiple unique[1] seeds may be advantageous if one wants to increase code coverage.

A seed should preferably be small. In the case of a mutational fuzzer, the size of the seed corresponds to how much data the fuzzer has to mutate. If possible, one

[1]Unique in the sense of the functionality that they trigger in the application.

should try to minimise the seed to its smallest possible size that does not affect the functionality it triggers. For example, fuzzing an image with 1920x1080 pixels will probably take a longer time than say 10x10 pixels, even tough the image trigger the same functionality in the target application, only with a smaller set of image data.

## 2.3 Fuzzing Framework

A fuzzer is not very useful without a framework (also called harness) that feeds the fuzzed output as input to a target application and observes its behaviour. The framework can also extend the fuzzer with different metrics that help the fuzzer decide if the current mutation or generation is beneficial [6, 2]. What is beneficial (or not) depends on the framework and what *observational power* it has, i.e. what it can learn from the executions of the target. The fuzzing framework can be a standalone application, or it can be built into the fuzzer itself. Whether it is a standalone application or not, a fuzzing framework will have the same basic observational power.

Observational power refers to what the fuzzer can observe of the target application during or, after execution. On the *nix platform and for locally running applications, the framework can observe signals sent to the target, such as `SIGSEGV`, `SIGABRT`, or `SIGFPE`; the (non-zero) return code; and timeouts whose length is specified in the framework. These are the basic observational powers that all fuzzing frameworks will have access to.

Signals are used for inter-process communication to notify a process of different events. The signals can be generated by the process itself, another process, or the operating system kernel. Upon receiving a signal, the receiving process will be interrupted by the operating system and its signal handler will be executed. If the process previously have not defined a custom signal handler the default handler will be used. Depending on what signal is received and the configuration of the process it may ignore the signal; use its own signal handler; terminate execution; or terminate and generate a core dump [16]. An application that terminated due to a received signal will have a return code equal to the unique signal identification number.

### 2.3.1 Feedback Loop

A feedback loop allows the fuzzer to make an informed decision whether a seed is useful. A fuzzing framework will monitor the target application in order to discover if the generated or mutated seed led to a new and previously undiscovered path in the program. Figure 2.1 visualises this concept, where feedback is provided from the target application to the fuzzing framework which then can make informed decisions about the usefulness of the generated or mutated data. Both generational and mutational fuzzing frameworks make use of feedback loops.

Some fuzzing frameworks allows the user to write custom feedback loops, that are used alongside the ones built into the framework. This expanded expressiveness allows the user to provide custom feedback or trigger action(s) when some expected

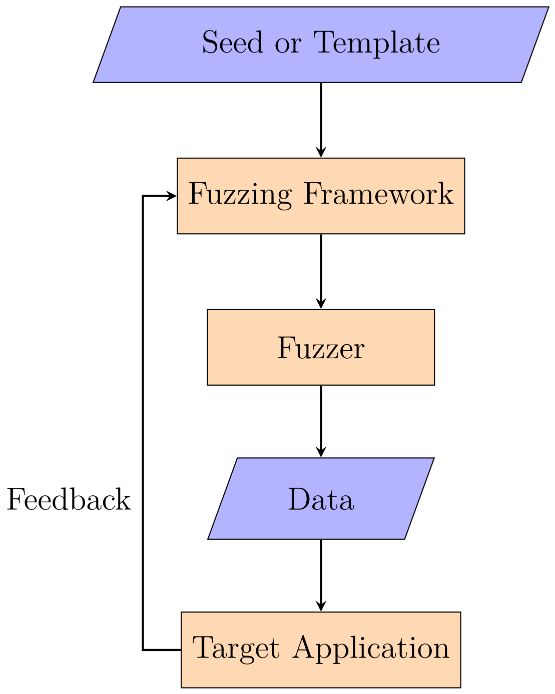


**Figure 2.1:** Fuzzing process where a feedback loop is employed

behaviour in the target application occurs.

### 2.3.2 Distributed Fuzzing

The execution speed of a fuzzer is important. Greater computational power enables faster fuzzing of the target. One way of achieving a higher execution speed is to distribute the workload. This can be done by running fuzzers on different servers and syncing the work. The framework can help with this, as well as telling the fuzzer to divide its technique in order to avoid doing the same work on two different machines. This obviously depends of the fuzzer used, since some fuzzers use a stochastic approach while others use a more deterministic approach to the mutations or generations made.

### 2.3.3 Crash Identification

Identifying, bucketing, and minimising bugs is a major challenge when fuzzing. Once an input that causes a crash has been identified. The framework needs to determine if another input has caused the same crash. It also needs do bucketing and sort the crashes depending on what signal caused it. For instance, a `SIGABRT` is probably not as interesting as a `SIGSEGV`.

For each crashing test case, not every bit field is relevant for the crash. It is possible to try and revert the seed as much as possible towards the original. By doing this you get the minimal case that still cause the same crash [17]. There also exist research on other ways of identifying bugs, that for example compares program state and stack pointers [18, 19]. Many fuzzing frameworks include tools for helping with these tasks.

# 3
# Fuzzer Evaluation

This chapter details the evaluation process of the two selected fuzzers, AFL and Honggfuzz. It presents the criteria for the selection of fuzzers, a detailed analysis of both fuzzers' capabilities, the target applications, and lastly the results of the fuzzing process.

## 3.1 Criteria

In order to make the fuzzer evaluation interesting and up-to-date we settled on the following criteria:

1. Open source and publicly available.
2. Mature and up-to-date.
3. State-of-the-art.
4. Target and run on the *nix platform.
5. Include, or have a built-in, fuzzing framework.
6. More advanced observational powers than the rudimentary observational powers described in Section 2.3.

### 3.1.1 Rejected Fuzzers

There exist many different fuzzers, some purpose-built, some more general purpose. Below are some common and interesting fuzzers listed which were considered but were not accepted for further study.

**Radamsa** is a multi-purpose mutation based fuzzer written in Scheme Implementation (SCM). Radamsa is a fuzzer, without any built-in framework, that mutates data through stdin. The data is mutated in a non-deterministic fashion with the option of providing a seed value to the random function. Due to the fact that Radamsa does not include a fuzzing framework, the tester will have to take care of possible error detection and crash handling themselves, but is nevertheless useful in conjunction with other fuzzing frameworks. Radamsa has been used to find security related issues in Mozilla Firefox and Google Chrome among others [20].

**Sulley** is a generational fuzzing framework written in Python [21], targeted towards network protocols. However, Sulley can also target file-formats and normal file consumer applications. We did not study this fuzzer due to not having advanced observational powers and having to specify our own templates introduces bias and allows for the fuzzers to not being objectively evaluated.

**Peach** is a generational and mutational fuzzing framework written in Python [22]. Uses so called "Peach-pits" for template creation, that, while being very expressive, takes time to implement correctly. We decided not to study this fuzzer due to the community version of Peach not having received an update for a long time and the commercial version not being open source and publicly available.

**Zzuf** is a mutational fuzzer with limited fuzzing framework capabilities [23]. Zzuf works by intercepting certain function calls and fuzzing the output that is sent to the target application. We decided not to study Zzuf due to it not having advanced powers of observation.

**QuickFuzz** is a Haskell built fuzzer that uses an interesting approach to generational based fuzzing. Normally generational based fuzzers needs someone to manually specify the template used. What QuickFuzz does instead, is leverage Haskell's type system as a grammar template using libraries from Hackage [24]. This approach has the advantage of not having to manually specifying the template, but with the disadvantage of losing expressiveness in the template. We decided not to study this fuzzer due to not having advanced observational powers and it being considered somewhat experimental.

## 3.2 American Fuzzy Lop (AFL)

AFL is a deterministic mutation-based fuzzing framework that uses injected code instrumentation to detect branching in the compiled binary. If a seed triggers new branches in the target application AFL considers this mutation interesting and will add it to the input corpus. This simple approach means that AFL is very good at testing large portions of the source code. AFL is a very successful fuzzer (in terms of security related bugs found) and has found bugs in security critical systems such as OpenSSH and GnuTLS , among others [6].

### 3.2.1 Instrumentation

Injected instrumentation is a way for AFL to detect the branch coverage for each test case. This allows it to make decisions about the usefulness of that particular test case. The instrumentation is code that is injected at each branch point in the target application. This code updates a memory map that is shared between the target and AFL with information about the branch taken [15].

AFL's use of instrumentation means that access to the source code is necessary and that the target binary have to be compiled with a compiler wrapper. AFL ships with wrapper around GCC and Clang, the two most common compilers on the UNIX-platform, that is responsible for injecting the instrumentation during the assembly phase of the compilation. There also exists a mode for fuzzing target binaries where the source code is not available. This mode is, however, two to five times slower than the normal injected instrumentation [25].

At the time of this writing, there exists an experimental LLVM mode for AFL that uses compiler level instrumentation instead of the assembly injection that is done in normally done. This allows for features that is intended to greatly speed up the fuzzing process. One of these features is the *deferred initialisation*. By inserting `__AFL_INIT()` in the source code, AFL's forkserver is able to clone the process at that point. This considerably reduces the overhead of creating new processes and since the code snipped can be placed after the initialisation of expensive resources, the speed can be increased further. However, care has to be taken when inserting this snippet since by placing it wrong, the binary may stop working as intended.

### 3.2.2 Crash detection

AFL uses a primitive crash detection technique. When the target application is stopped for some reason, AFL will look at the return code and the signal that caused the target to stop (if applicable). The crash will be bucketed with respect to the signal that caused the crash or hang.

In combination with the instrumentation of the target, AFL is able to determine whether a crash is unique. If the path taken leading up to a crash is unique, the crash is also labelled as unique. This means that there can be multiple test cases classified as unique for the same bug. This is intentional since when the bug causing the crash has been fixed, the crashing test cases can be used for regression testing, thus testing all paths leading to the (hopefully) fixed bug.

## 3.3 Honggfuzz

Honggfuzz is a non-deterministic mutation-based fuzzer that employs the Linux kernel subsystem *Perf* in order to determine successful mutations [26]. It does this by checking the code coverage induced by the seed. On subsequent mutations, seeds are only added to the input corpus when a new mutation has a larger coverage than the previous. Honggfuzz has a built-in framework with support for using an external fuzzer instead of the built-in one.

### 3.3.1 Crash detection

Process trace (`ptrace`) is used by Honggfuzz to observe the behaviour of the target application. `Ptrace` is a debugging tool that allows a process to inspect and control the execution of other processes. `Ptrace` attaches to a target application, referred to as a *tracee*, and allows a *tracer*, e.g. a fuzzer, to view registers and memory of the tracee.

When a signal is sent to the tracee, `ptrace` will stop the execution, thus allowing the tracer (Honggfuzz) to inspect the target application. Depending on the signal sent, Honggfuzz can make the decision to either terminate the execution or continue it. If the target application received a crashing signal (`SIGSEGV`, `SIGABRT`, `SIGFPE`, etc), Honggfuzz will save the input and bucket it with respect to the current program counter, instruction that caused the crash, and the signal received. Information such as stack-pointer address is also saved.

### 3.3.2 The Perf subsystem

Modern CPU have dedicated built-in registers that monitor performance factors, so-called hardware performance counters [27]. Perf is a kernel-based subsystem performance analysis tool for Linux that provides an interface to these counters [28]. Perf allows for statistical analysis on hardware level components but also software components. At the hardware level, the performance counters observe events such as number of instructions and branches. The number of instructions or branches executed are useful in fuzzing for determining if the path of execution changed during two different executions of the target application. Since the counters are implemented in hardware, the usage of Perf will have a low impact on performance and is faster than code instrumentation [29].

It is important to note that Perf provides a bird's-eye view of an execution and not in-depth knowledge of the path taken within a program and results will vary to a degree for each execution. Listing 2 provides a simple example how this can become an issue, the same amount of work will be performed whether the first or second branch is taken in the program. The statistical measurements will therefore provide the same results for both branches, whilst an instrumented approach could identify which of the different branches were actually taken during the execution.

```
chance = flipCoin()
if chance > 0.5 then
    // First branch
    print "A"
else
    // Second branch
    print "B"
exit(0)
```

**Listing 2:** Example where Perf cannot distinguish the branch taken during execution.

## 3.4 Target Applications

American Fuzzy Lop and Honggfuzz were tested against a set of applications. We choose to use a subset of the DARPA Cyber Grand Challenge (CGC) target suite [30]. The suite consists of over 100 programs (called challenges) written in C that takes input through stdin. The challenges have been specifically designed to test automated security vulnerability finding tools. Each of the challenges have one or more vulnerabilities of varying difficulty and type. All vulnerabilities are defined and described in the documentation of each program. Since the target suite has been used in the CGC competition, and thus has been extensively tested, it is unlikely to contain unknown bugs or vulnerabilities.

A custom-made operating system was built for the CGC competition to ensure higher reproducibility of bugs and easier implementation of automated testing tools [31]. The kernel used in the operating system has a reduced set of only 7 system calls. As a side effect, the target suite per default is not POSIX compliant, and will not work on other platforms without modifying the source code. To enable compatibility with the selected fuzzers, and keep changes of the challenges to a minimum, we built a wrapper that translates the custom system calls to POSIX standard allowing the target applications to run on a normal Linux system. Furthermore, the platform does not utilise the standard C library (libc), hence many standard functions in C are not available and have been reimplemented for each individual challenge that needs them. Due to the use of a global namespace in C, the CGC reimplemented libc functions were removed and replaced with the original ones from the standard C library. Examples of this includes the functions `strlen()`, `toupper()` and `atoi()`, etc.

The results of this fuzzer evaluation is used as a baseline for the anti-fuzzing algorithms tests presented in Chapter 4 and 5.

### 3.4.1 Cyber Grand Challenges

Due to temporal restrictions, we choose not not to convert all CGC challenges, but instead focus on the preselected example set. The example set consists of 17 programs that each have different challenges and vulnerabilities.

Some of the challenges were removed from the fuzzer evaluation. The reasoning for removing the challenges varies, such as them not being applicable on the fuzzers selected for evaluation or no longer being vulnerable after the change to POSIX standard.

We describe below the example targets that we choose to use in our evaluation of AFL and Honggfuzz. The seeds that was provided to the fuzzers when performing the tests can be found in Appendix A.1.

### CADET00001

CADET00001 is a palindrome checker. Upon starting the program, the user is prompted to input a text. The program will then validate check if the supplied input is a valid palindrome. The program will repeatedly query the user for more input until an `EOF` is received or the program is terminated in some other manor.

**Vulnerability** The program uses a buffer for storing the input provided by the user, which is 128 bytes long. Therefore, the application is vulnerable to a stack-based buffer overflow attack if the provided input is larger than 128 bytes.

**Difficulty (Easy)** A fuzzer would have to produce an input value larger than 128 bytes in order to find the vulnerability. Due to the simplistic nature of this program, it was rated as being easy.

### CADET00003 (Removed)

CADET00003 contains the exact same functionality as CADET00001.

**Reason for Removal** The majority of the code of CADET00003 is the same code as CADET00001, with the exception of the name of a single `#define`. Due to the fact that it is essentially a duplicate of another challenge it was removed from the set of targets. Running this challenge would cause duplicate results.

### CROMU00007 (Removed)

CROMU00007 is a payroll system in which the user can register employees, their salaries and working hours. Once information have been registered with the system it is possible to make queries on how much employees have earned.

**Reason for Removal** The crash was not reproducible after the migration from the CGC platform.

### EAGLE00004 (Removed)

EAGLE00004 is a simple calculator-like program with very limited functionality. It consists of three separate programs which are to be connected through pipes, program one will send information to program two, which in turn will send information to program three.

**Reason for Removal** Since the target consists of 3 different programs that use inter-process communication they are not applicable on the fuzzers selected without major modifications to the source code. To keep the result as unbiased as possible the target was removed.

**EAGLE00005**

EAGLE00005 is a interactive hangman game. To be able to play the game the player must first enter a password, "HANGEMHIGH!", and then select a 4 letter seed which determines the secret word for the game. The user is then prompted to enter one letter at a time until he completes the word or runs out of chances. If the user completes the game he will be able to enter a name into the list of highscores else-wise the game will exit.

**Vulnerability** A stack-based overflow vulnerability is located in the highscore list functionality. When the user enters his name to the highscore it is possible to cause a buffer to overflow.

**Difficulty (Medium)** In many ways EAGLE00005 is similar to CADET00001, they both contain a simple buffer overflow vulnerability. However, to access the vulnerability in EAGLE00005 the game must b completed first. The game itself is not trivial but far from impossible. Therefore this target was rated as being of medium difficulty.

**KPRCA00001**

KPRCA00001 represents a simple protocol. The user must initiate the authentication process by calling `HELLO`. The service will respond with a message in the format `OK <code>`, where `<code>` is a randomly generated one-time password of length 8 containing uppercase letters and numbers. This password is then to be repeated by the user with the command `AUTH <code>` back to the service in-order to authenticate. Once authenticated the user can set data and issue calls to different parts of the service.

**Vulnerability** The program contains a stack-based overflow when calling a specific command to send data to `/root64`. To trigger the vulnerability, the user must do a series of steps: authenticate, set the mode to `encode`, set data to a string larger then 64 bytes, and call `/root64`.

**Difficulty (Hard)** The program was classified as hard because of the authentication process. It is hard for a fuzzer to guess a random value, and since the code is randomly generated for each execution it cannot be provided as a seed.

**KPRCA00003**

KPRCA00003 is an image compressor. The service takes a bitmap with maximum size of 256x256 pixels as input and outputs a 128KB image in the ACGC image format.

**Vulnerability** In order to make the program crash, a bitmap supplied needs to be close to, but not larger than, 256x256 pixels and requested to be converted to a encoding quality of near 100. This will result in the output file being larger than 128KB and therefore overflowing the output buffer.

**Difficulty (Hard)** A fuzzer must first create a valid bitmap, and then fit the needs to crash the requirements for the program to crash. The combination of many hard elements to reach the vulnerability we rated this target as being hard.

### KPRCA00015

KPRCA00015 implements a RFID-protocol for wireless transactions. The program can for example issue new cards, perform purchases, perform refunds, display card balance, and display transaction history.

**Vulnerability** A vulnerability is located in the refund functionality. To issue a crash the user must first create a card and authenticate himself with that card, make transactions and finally issue a refund which is vulnerable. To enable the vulnerability, an Untrusted Pointer Dereference[1], the reference to the transaction must assert values that allows it to bypass the verification process used in the refund command.

**Difficulty (Hard)** To trigger this vulnerability requires the program to go through a series of steps and perform an authentication process. Furthermore, there are verification and validation processes used within the program that hardens it and lowers the probability of a fuzzer to reach the vulnerable code. We estimate that the vulnerability to be very hard for a fuzzer to trigger.

### LUNGE00002

LUNGE00002 provides a lookup service with support for dynamic skip searching, prefix tree search and naive compression.

**Vulnerability** The program contains a out-of-bounds memory corruption vulnerability that is triggered when a specially crafted entry is added to the lookup service and a search is performed. When performing the search the newly added entry will cause the program to try and locate a delimiter outside its buffer consequently leading to the corrupting memory.

**Difficulty (Medium)** The program takes simple commands, such as `ch_sec` and `make_sec` for changing and making sections. Without proper seeding it would be hard for a fuzzer to find these unique strings. However, with the commands and the delimiter supplied in the seed or through the use of a dictionary a fuzzer would have an easier time. To make the program crash a fuzzer would have to use the commands and delimiter combined with random data in a manner that creates a bad formatted entry and then try to access it.

### LUNGE00005 (Removed)

LUNGE00005 consists of 6 small programs, each providing a different functionality; command and control, contains, does-not-contain, word-count, compression and de-

[1]`https://cwe.mitre.org/data/definitions/822.html`

compression. By connecting the programs and sending information it is possible to trigger an overflow vulnerability in the decompression program. However, to achieve this the programs must be run in a certain order and with specific input.

**Reason for Removal** Since LUNGE00005 make use of inter-process communication it is not applicable to the fuzzers chosen for evaluation.

### NRFIN00003

NRFIN00003 is a HTTP-like service that allows for a user to manage his internet connection. The user may execute a command to upgrade his subscription in order get access to the faster speeds. However, to do so the user must have enough credits to afford it.

**Reason for Removal** The vulnerability was not reproducible after converting the application from the platform.

### NRFIN00010

NRFIN00010 is a transaction storage and processing environment for RFID card payments. Within the system transactions are performed in two or three steps of the following steps; Initiation, Authentication, Operation and Finalisation. The program supports functions such as purchases, refunds, display listings of transaction history, balance inquirers and more.

**Vulnerability** NRFIN00010 contains three null pointer dereference vulnerabilities. The vulnerabilities are very similar and is caused due to transactions not being atomic. For example, when performing a refund the reference to a purchase is removed from a transaction during the Operation step. If another command, e.g. listing of transaction history, is issued before the the finalisation step is performed a null pointer dereference.

**Difficulty (Hard)** There is a wide range of commands that can be executed in this program and they require input to be formatted in a specific manner. Generating the commands for the input would probably be hard and performing several commands in the correct order even harder.

### NRFIN00013

NRFIN00013 is a calculator intended to apply mathematical operations on lists of numbers. The calculator can perform numerous tasks such as evaluate additions, average, max, min, sort and much more. In order to use the calculator the user must first enter a nonce, a randomly generated number only used once, which is outputted upon starting the program.

**Vulnerability** The program contains more than 15 vulnerabilities and 6 of them may result in a crash, such as `SIGSEGV`, `SIGFPE` or `SIGALRM`. Many of the vulnerabilities are possible due to improper input validation.

**Difficulty (Hard)** The use of a nonce in the beginning of this program makes it hard for a fuzzer to get deep within the program. Bypassing the initial step is unlikely due to the fact that random values are being generated.

#### TNETS00002

TNETS00002 is a program that allows a user to have virtual pets. Pets can be created, deleted and perform simple commands.

**Vulnerability** TNETS00002 contains a heap based buffer overflow. If the user renames a pet after it's creating the new name gets appended to the old one. If the name is changed several times this will cause a buffer to overflow.

**Difficulty (Hard)** In order to trigger a crash the fuzzer must first create a bet, rename it with a long name, several times and then issue a command to the pet. Each action must also be performed with a sequence number. Due to the many steps with advanced input this program was rated as hard.

#### YAN0100001

YAN0100001 is a game similar to Battleship. Each player place their ship and then take turn to sink the other players ship.

**Vulnerability** When each command is executed it is stored in a 512 byte buffer. If a command entered is longer than 512 bytes a buffer overflow will occur.

**Difficulty (Easy)** To find the vulnerability a fuzzer would have to produce a value large enough to overflow the buffer, the challenge was rated as easy.

#### YAN0100002

YAN0100002 is a calculator for the trajectory of a tennis ball. The user is prompted to input the initial velocity in both X and Y direction, and initial count (when in time to start evaluate the position of the ball).

**Vulnerability** Due to several insecure type conversions and usage of floats the program will misbehave under certain conditions. If a the initial count is set to a large value some evaluations will be performed incorrectly that may leads to several vulnerabilities, and finally a out of bounds array write. In total the source code contains three types of vulnerabilities: Incorrect Calculations[2], Incorrect Conversions between Types[3] and Out of Bounds Write[4].

**Difficulty (Medium)** For the program to crash several values has to be set. However, as explained in Section 2.1.2.1, many fuzzers make use of *Interesting*

[2]http://cwe.mitre.org/data/definitions/682.html
[3]http://cwe.mitre.org/data/definitions/681.html
[4]http://cwe.mitre.org/data/definitions/787.html

*values* that are likely to cause crashes, including extreme values. Therefore increasing the chance of a bug being triggered.

#### YAN0100003

YAN0100003 is a simple character counter. The program takes reads a file and then will return the number of special characters, numbers, spaces, non-printable characters, lower and uppercase letters there are in the file.

**Vulnerability** The program contains a *out of bounds read* because of an 32-bit unsigned integer operation are made on an 8-bit unsigned integer. If the input contains a carefully selected number of spaces the usage of the operation may lead to a segmentation fault due to the adjacent memory being a pointer.

**Difficulty (Easy)** Like the other targets rated as easy this one requires the fuzzer to produce large amount of data for it crash. The vulnerability is of a simple nature, and if the fuzzer manages to produce input with enough spaces the program will crash.

### 3.4.2 MediaInfo

A "real-world" program was also selected for testing to achieve a more complete view of the fuzzers' performance. MediaInfo is a program that displays metadata information about the file supplied [32]. This is information such as width and height if an image is supplied or bitrate, length etc if it is a video. MediaInfo has support for a large number of different file formats, making it an ideal fuzzer target.

MediaInfo is different from the CGC challenges. It was initially released 2002 and since then it has grown to a size of more than 100 000 lines of codes. The project have mainly been written in C++ which makes it susceptible to several types of attack if the code have not been written properly. Because of the support for multiple different file-formats, the parser will inherently be complex, resulting in an enormous amount of paths that may be taken during executing of the program.

## 3.5 Evaluation Technique

Ultimately, the goal of a fuzzer is to find vulnerabilities. One way vulnerabilities may reveal themselves is through crashes. Therefore crashes found with respect to time is a vital measurement, but factors such as poor crash identification and bucketing may lead to duplicate bugs which interferes with the accuracy of the result. To ensure an accurate measurement, the bugs identified were verified crashing and verified unique to the best of our ability.

Each target were fuzzed for 24 hours with both Honggfuzz and AFL. Honggfuzz used 16 threads while AFL used 16 processes, each locked to a separate CPU core. AFL were run in the parallel mode, meaning that the master process are run normally, whilst the slaves are skipping the deterministic steps.

### 3.5.1 Testing Platform

All tests were performed on a dedicated server running the latest, at the time of this writing, long term support server version of Ubuntu, 14.04.04 LTS. The server has 8 cores and 96GiB ram. The full specification of the testing platform can be seen in Table A.2.

## 3.6 Result

This section details the results of over 900 hours of fuzzing and some 84 000 cumulatively reported crashes.

### 3.6.1 Cyber Grand Challenges

Each target was seeded with suitable data, the full specification of seeds can be seen in Table A.1.

Out of the twelve CGC targets fuzzed, crashes were detected in five of them. An overview of the targets that exhibited crashes can be seen in Table 3.1. Both Honggfuzz and AFL managed to find crashes in the challenges CADET00001 and YAN0100001. These two challenges are very simple buffer overflow attacks, where large input directly causes a crash. Furthermore, AFL was also able to find crashes in EAGLE00005, NRFIN00010 and YAN0100003. We rated these challenges as being of easy to medium difficulty.

An interesting note is that both fuzzers were unable to find any vulnerabilities in the programs that were rated hard. These programs contained protection mechanisms, such as an initial challenge-response step, than mutational fuzzer will have trouble with. Re-running these targets with a dictionary supplied to the fuzzers would possibly have improved the probability of finding crashes. We leave this as possible future work.

We used a wrapper script around GDB that when fed a location of crashing inputs and a target, will compare the stack trace of each crashing input and prune those that have the same backtrace. The full script can be found in Appendix A.4. Out of the 83 848 reported crashes by both fuzzers, 11 crashes were left after running the script. Table 3.2 show the full results after culling the crashing inputs. This set of test cases is more suitable for human evaluation in order to determine, for instance, exploitability.

Figure 3.1 show crashes reported by Honggfuzz and AFL over a 24 hour period. We can see that AFL found the majority of all reported crashes within the first hour, while Honggfuzz continuously reported crashes over the whole 24 hour period.

| Target | AFL | Honggfuzz |
|---|---|---|
| CADET00001 | ✓ | ✓ |
| EAGLE00005 | ✓ | ✗ |
| NRFIN00010 | ✓ | ✗ |
| YAN0100001 | ✓ | ✓ |
| YAN0100003 | ✓ | ✗ |
| KPRCA00001 | ✗ | ✗ |
| KPRCA00003 | ✗ | ✗ |
| KPRCA00015 | ✗ | ✗ |
| NRFIN00003 | ✗ | ✗ |
| NRFIN00013 | ✗ | ✗ |
| TNETS00002 | ✗ | ✗ |
| YAN0100002 | ✗ | ✗ |

**Table 3.1:** Table showing which of the targets emitted one or more crashing states for each of the selected fuzzers.

| Target | AFL (Reported) | AFL (Unique) | Honggfuzz (Reported) | Honggfuzz (Unique) |
|---|---|---|---|---|
| CADET00001 | 38 757 | 3 | 1 229 | 1 |
| EAGLE00005 | 40 893 | 2 | - | - |
| NRFIN00010 | 64 | 1 | - | - |
| YAN0100001 | 1 092 | 2 | 1 685 | 1 |
| YAN0100003 | 128 | 1 | - | - |

**Table 3.2:** Table showing crashes reported by each fuzzer as well as the number of crashes after running the crash pruning script from Appendix A.4.

AFL's behaviour is consistent with what is specified in the README file: "If a single bug can be reached in multiple ways, there will be some count inflation early in the process, but this should quickly taper off." [25].

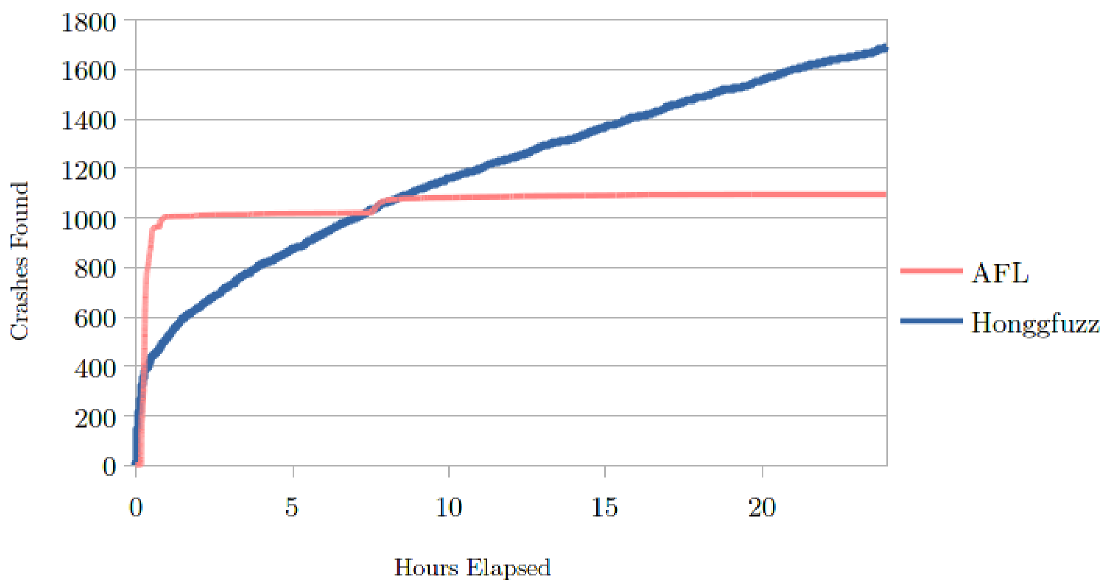


**Figure 3.1:** Crashes reported by AFL and Honggfuzz on YAN0100001.

#### 3.6.1.1 Execution speed

Figures 3.2 and 3.3 show the execution speed over time during a portion of the fuzzing of the target CADET00001. As can be seen, AFL reached an average cumulative execution speed of roughly 48 000 whilst Honggfuzz only achieved around 1 100 executions per second. The exact reason for the big difference in execution speed is unidentified, but several factors could play a role. Honggfuzz uses `ptrace` which needs to attach to process before execution; AFL instruments the code before execution; and the scalability of the fuzzer when using several cores.

### 3.6.2 MediaInfo

MediaInfo was fuzzed for 7 days with AFL and Honggfuzz, for a total of 14 days. The seeds consisted 24 different file formats, such as JPEG, PNG, OGG, etc. A full list of the seeds can be seen in Appendix A.2.

#### 3.6.2.1 Execution Speed

The overall executions per second is low for both fuzzers, coming in at around 25 execs/sec. We tried various solutions for trying to speed up the process, such as deferred initialisation, stripping symbols in the binary in order to reduce the binary size and removing all but the essential parts of the source code for the CLI, but

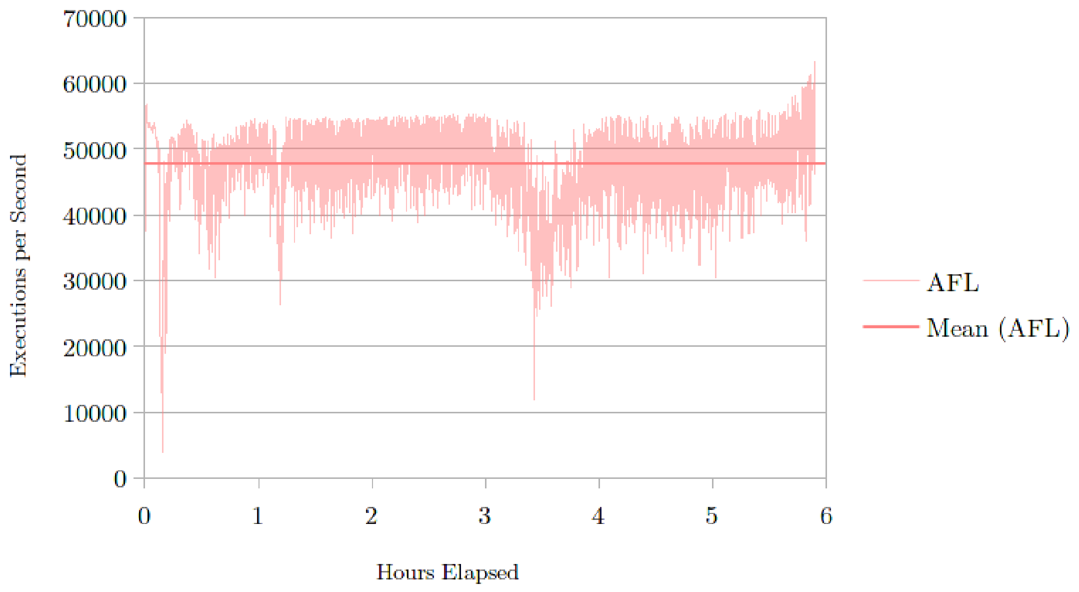


**Figure 3.2:** Graph showing the number executions per second for AFL on CADET00001.

| AFL (Reported) | AFL (Unique) | Honggfuzz (Reported) | Honggfuzz (Unique) |
|---|---|---|---|
| 9 624 | 20 | 8 | 2 |

**Table 3.3:** Table showing how many times AFL and Honggfuzz crashed MediaInfo as well as how many crashes were left after pruning.

it seems that MediaInfo is just slow and somewhat poorly optimised. The fuzzers are not the bottleneck with regards to execution speed. An execution speed over at least 100 would have been preferable.

#### 3.6.2.2 Crashes

The number of crashes reported by AFL and Honggfuzz can be seen in Table 3.3. It is important to note that the crashes reported by a fuzzer is not unique crashes, even though the fuzzers have reported so. As discussed in Section 3.2.2 and 3.3.1, Honggfuzz and AFL have different means of determining unique crashes and therefore also different definitions of what a unique crash is.

During the analysis of the generated crashes reported by AFL, it was discovered that AFL had reported several false-positives. The number of false-positives generated overtime can be seen in Figure 3.4. Crashes that we were able to reproduce during the crash analysis phase. A common factor among these false-positives was that they all were killed through the `SIGABRT` signal. Although the exact cause of these false-positives not been determined, it is likely that they are caused due to the stress put on the system. Honggfuzz, on the other hand, did not produce any

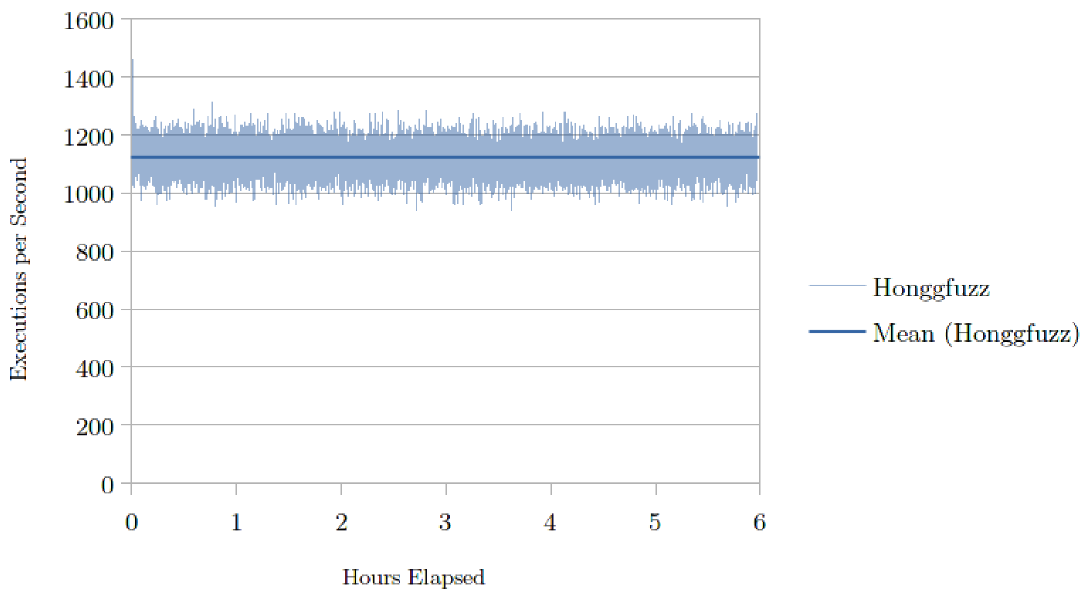


**Figure 3.3:** Graph showing the number executions per second for Honggfuzz on CADET00001.

false-positives.

#### 3.6.2.3 Resource Utilisation

During the testing of MediaInfo, the limitations of the testing platform was investigated. Figure 3.5 shows the CPU and memory of AFL and Honggfuzz during a 24 hour fuzzing process of MediaInfo. AFL utilised all the power of the CPU whilst Honggfuzz had an average of 44% CPU used.

It was discovered that Honggfuzz seems to have memory leakage issues. During the 24 hours of fuzzing Honggfuzz continuously kept increasing its memory usage and afterwards the memory was not released. These issues were apparent while fuzzing the other targets as well; the memory was only released upon restarting the testing platform.

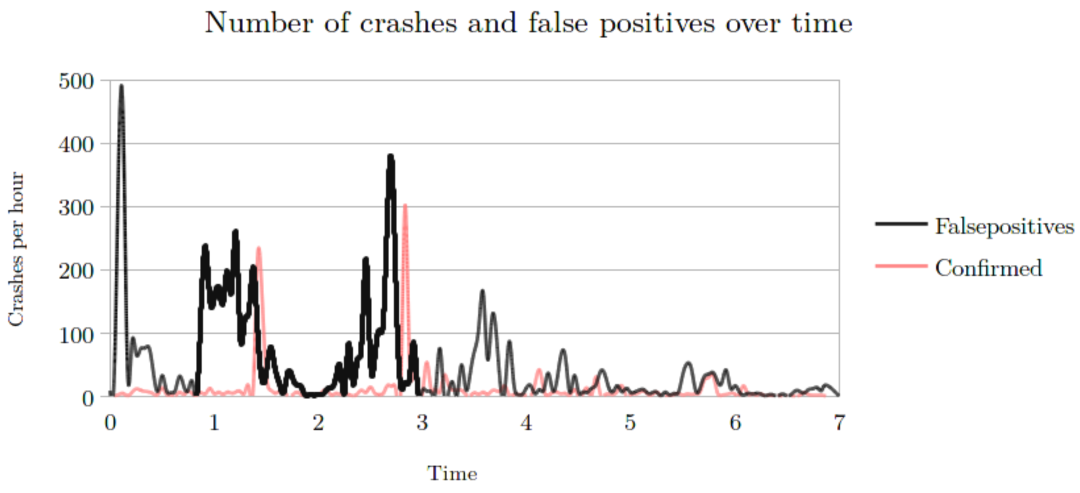


**Figure 3.4:** Graph showing the number of false positives generated per hour by AFL on MediaInfo.

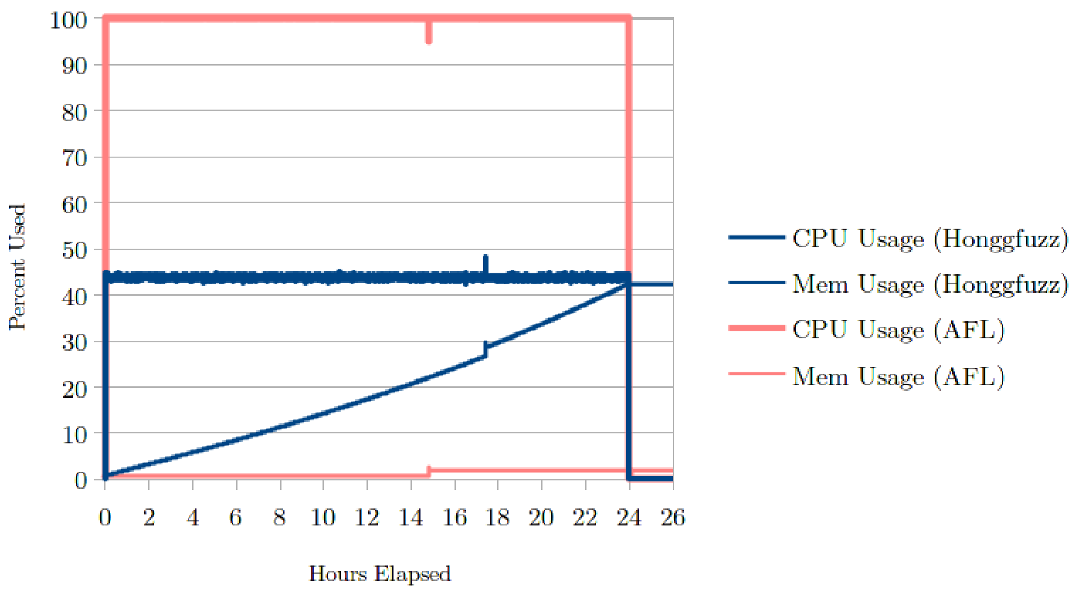


**Figure 3.5:** Graph showing the CPU and Memory usage of AFL and Honggfuzz when fuzzing MediaInfo for 24 hours.

# 4 Anti-Fuzzing

The process of hindering and/or delaying a fuzzer from effectively fuzzing a specific target is called *Anti-Fuzzing*. To date, little to no research has been done on the subject. In this chapter, the rationale behind anti-fuzzing is explained, the objective outlined, and the approach we used is detailed.

## 4.1 Rationale

One could imagine a use case for anti-fuzzing as follows: a company has developed a new application that they want to release as quickly as possible. This could mean that they will not have time to perform a proper security evaluation of their application before release (fuzzing a target could take months in some extreme cases). The aforementioned company could then develop and apply an anti-fuzzing algorithm to their newly released application. This would mean that any adversaries trying to fuzz the application would be delayed or maybe even completely hindered. All the while the company is doing internal fuzzing on their application, without the anti-fuzzing mechanics applied. Consequently, the software company will have a significant advantage over any external testers.

Anti-fuzzing is possibly a controversial subject. Our rationale for researching anti-fuzzing is to find possible improvements that can be applied to fuzzers in order to detect and warn the tester of anomalous behaviour. Another reason is to research what kind of anti-fuzzing measures which could be applied to a target application, and how much effort it is, to cripple or slow down state-of-the-art fuzzers.

Finding vulnerabilities in a target application can be a good thing or a bad thing, depending on who finds them and what their intentions are. Developing an anti-fuzzing algorithm will hinder all testers from fuzzing, no matter their motivations. In some sense, anti-fuzzing is security-by-obscurity since any action the anti-fuzzer takes to hinder the fuzzer, that does not affect normal users, will be avoidable.

## 4.2 Objective

An anti-fuzzing technique has the objective to prevent or delay the detection of crashes. The objective can be achieved by targeting the observational powers of a fuzzer. Generally speaking, a fuzzer has two types of observational power: crash detection, the means of detecting when the target crashes; and metrics, the means of generating feedback and information to the framework.

Depending on what observational powers a fuzzer has and how they are implemented, different approaches needs to be taken in order to fool it. Honggfuzz, for example, uses the Perf subsystem (described in 3.3.2) as a metric to determine the number of branches taken in a target, while AFL uses its injected instrumentation to do more or less the same thing.

## 4.3 Approaches

A program that has had anti-fuzzing techniques applied should, from a user perspective, have unchanged behaviour. However, when the same application is run through a fuzzer, it is allowed to behave differently depending on what techniques have been applied. The techniques applied can be broken down into two major categories: active techniques that modifies the behaviour or metrics only when the target is being actively fuzzed; and, passive techniques that are "always on" and have the same behaviour regardless of the target application being executed normally or through a fuzzer.

### 4.3.1 Active

An *active* anti-fuzzing technique only applies upon detecting that the target application is being fuzzed. While the target is running normally, the anti-fuzzing techniques lie dormant and unused. An active anti-fuzzing approach works in a fashion similar to anti-virus software. Each "virus" (fuzzer) needs to be fingerprinted beforehand in order to be detectable. There does not exist any catch-all solution for detecting any and all fuzzers. For example, a simple way of implementing an active anti-fuzzing technique would be to look at all running processes in the system. If a known fuzzer is running, then the program could assume that it is being fuzzed, and execute some measure designed to, for example, falsify any metrics that the fuzzer is subscribing to. This may obviously produce false positives and is an unreliable method.

Considering that the active anti-fuzzing approach require the target application to actively detect it is being fuzzed, there will always exist solutions where the detection algorithm are bypassed. In the previous example, where the target application compared running processes against known fuzzers, the tester can circumvent the check as easily as renaming the fuzzer process. To counter this, one might imagine that the target application can do a hash of each running process and compare against a known set of hashes. This approach is obviously extremely costly and not very scalable since it would require the target to know about each and every version of all fuzzers it wants to target. Countering this approach is again, very easy. A minor modification will produce a different (unknown) hash.

A different approach for target applications to detect that they are being fuzzed is to try to detect signature techniques. A fuzzer may affect the running target application in some subtle way that is detectable. For example, in the case of `ptrace`, a tracee may be able to detect it is being traced. However, if a fuzzer does not affect the target application, then detection will be harder.

Assuming that the fuzzing detection technique is reliable, i.e. will detect all fuzzers and produce no false-positives, an active anti-fuzzing approach has the benefit of not affecting normal end users whatsoever. The anti-fuzzing mechanism will only be activated once fuzzing occurs and thus from user perspective it will still have the same behaviour. This may also harden the ability for a tester to analyse the program since it will behave differently when the tester runs the program as a user compared to when it is being run by the fuzzer.

### 4.3.2 Passive

A *passive* anti-fuzzing technique will always be performed. It may target the observational powers of a fuzzer or aim to lower the performance by reducing maximum number of executions per second possible, as described in section 2.1, a high execution speed is crucial when fuzzing. From a user perspective, a pause of a fraction of a second will hardly be noticeable, but for a fuzzer trying to execute a target as fast as possible the impact of the performance will be significant.

The negative aspect of a passive technique is of course that it will also affect normal users. For example, if crashes are masked or hidden, then if a user accidentally trigger a crash the behaviour of the application will be unexpected and possibly frustrating.

## 4.4 Anti-Fuzzing Techniques

During our research we identified four different attack-vectors against fuzzing harnesses. Note that since we only target mutational fuzzers, some of these techniques may differ for generational fuzzers.

**Execution Speed** By intentionally decreasing the performance of the target application, the fuzzer will not be able to reach a high execution speed. This can be done by accessing limited resources or making the application heavier, i.e utilising more CPU or Memory. This method will only delay finding any potential crashes and not hinder them completely. Given enough time a fuzzer may find them.

**Masking Crashes** The application may also choose to handle crashes internally, thus not revealing to the fuzzer the fact that it has crashed. This can be implemented through custom signal handlers for example.

**Detection** If the target application is able to detect the fuzzer it can choose act differently. A possibility is to simply exit the application directly when a fuzzer is detected.

**The Fuzzing Framework** Most advanced fuzzers often contain a framework with a feedback loop. The feedback is generated from observation of the target application during, or after a fuzzing run. With this in mind, a target application may use this knowledge to its advantage. For example a target application may be designed to exploit limitations of the framework or manipulate the metrics shown in the user interface, and thus not only fooling the fuzzer, but also fooling the user of the fuzzer. A target application may also exploit how the framework behaves in order to starve the computer of resources. For example, by making each and every generated test case interesting, the whole process fuzzing process is slowed down and eventually all available inodes will be exhausted.

An application may choose to implement one or several anti-fuzzing mechanisms and combine them in any order to get the desired result. For example an application may try to detect fuzzers and only if detected it will reduce the performance of the application, thus not affecting end users in any way.

To delay the fuzzing process as much as possible it is preferable if the techniques implemented are hard to detect. Consider the case where the binary hides all crashes from the fuzzer, if a fuzz tester does not discover the technique, the whole fuzzing process will be performed in vain. Each implementation of any anti-fuzzing technique will leave traces. What side effects a technique have will determine how easy it is to detect by an adversary. Therefore, to be aware of the toolbox that can be used by the adversary, such as `strace`, `strings` and general process information from `/proc/` is important when implementing the anti-fuzzing techniques.

An alternative approach is to make the binary resistant to reverse engineering. This forces the adversary to shift focus to trying to defeat the reverse engineering protection mechanisms or modify the fuzzer so it circumvents the anti-fuzzing techniques. Either way this raises the bar for the adversary making it substantially harder to correctly fuzz a target.

# 5

# Anti-Fuzzing Evaluation

This chapter presents the results and evaluation of our anti-fuzz testing.

## 5.1 Applicability

Applying an anti-fuzzing technique into a project will require adding additional source code. Hiding anti-fuzzing techniques in open-source projects is likely hard, and could easily be detected and removed. However, in closed source products it is not possible for an adversary to do code review and thus anti-fuzzing techniques are harder to detect and easier to hide. For our evaluation, we had access to the source code of all target applications and could therefore gain advantages not available to an adversary. Such an advantage is the instrumentation mode of AFL which would not be available without access to the source code.

Utilising anti-fuzzing comes with a risk. If discovered, the techniques might be countered or removed. Moreover, it may be hard to update the anti-fuzzing techniques applied to a binary after it is released. Anti-fuzzing is therefore not suitable as a long term solution for hiding vulnerabilities and should be viewed as a stop-gap solution when little time has been available for internal fuzzing of the product.

## 5.2 General Approaches

A general approach to anti-fuzzing is to switch out the fuzzer-supplied input to a known safe input when the application detects it is being fuzzed. A pseudo-code example of this can be seen in Listing 3. This has the advantage of running the application normally even when the target is being fuzzed, but without the possibility of crashing due to user input. Metrics will be largely unaffected and from the observers perspective, everything will look normal. Of course, always using the same input will, depending on the fuzzing framework in use, affect metrics. AFL for example, will never report any new unique paths reached. Switching the input to a known safe input will shift the focus of anti-fuzzing from using specific ways to fool a fuzzer to detecting the fuzzing process. Failing to detect the fuzzer will in that case effectively disable the anti-fuzzing technique. Instead of finding different ways of fooling a fuzzer, this approach only require the application to detect it is being fuzzed.

```
int main(argv)
    if(beingFuzzed?()){
        // Replace input with a randomly or deterministically selected safe input.
        input = getSafeInput()
    } else {
        input = getInput(argv)
    }

    return runProgram(input)
}
```

**Listing 3:** A simple wrapper of an program. Upon detection of a fuzzer any input will be replaced with a random safe input.

## 5.3 AFL

AFL uses a *fork-server* in order to increase performance by not having to make an excessive amount of `execve(...)` calls [33]. The fork-server is injected into the target application by AFL's compiler and is run before `main(...)` function. Upon receiving go-ahead from AFL, the fork server will fork itself: the child will run the normal target application code; whereas the parent has the responsibility to send status information to AFL when the child has exited. Upon the termination of the child process, the fork server will wait for the next go-ahead from AFL and the cycle will continue.

AFL does not use any advanced crash detection mechanisms, as described in Section 3.2.2. When the target has exited for some reason and the fork-server has reported the return values, the value is then used to determine whether a crash has occurred. If the return code is non-zero, then it will use macros defined in libc to decode more information about what happened to the target. This approach can be seen in Listing 4. `WIFSIGNALED` is a macro that will return true if a signal that was not handled were used to kill the process and `WTERMSIG` is macro that will return the signal identification number of the signal that terminated the process in question.

When the technique for detecting crashes is known it becomes easier to hide them. Since AFL uses a somewhat primitive technique for detecting crashes, the technique for masking them becomes somewhat primitive as well. We found two ways of masking crashes from AFL: the first is for the target application to `fork ()` and run the actual program in the child. Any signal sent to the child is not propagated up and detected by AFL; the second technique is to use signal handlers to intercept interesting signals such as `SIGSEGV`.

### 5.3.1 Masking Signals

The first technique were successful at masking crashes from AFL and since the program is run normally, the metrics are largely unaffected. From the fuzzer's perspective it looks like the program is being fuzzed normally since it is; the crashes are just not seen by AFL. However, due to some overhead when forking, executions per second is somewhat lowered using this method. It also seems that at least some crashes are interpreted as hangs. This trend is visible in Figure 5.1. The reason for AFL interpreting some crashes as hangs are unknown, but we theorise that it is due to AFL having a modest timeout value before considering a test case as a hang, and that when a `SIGSEGV` signal is generated and sent to the target application, that this process takes longer than the defined timeout value.

```c
if (WIFSIGNALED(status) && !stop_soon) {
    kill_signal = WTERMSIG(status);
    return FAULT_CRASH;
}
```

**Listing 4:** How AFL detects crashes in target applications

```c
int main(int argc, char const* argv[]) {
    pid_t child = fork();
    if(child < 0) {
        printf("Fork failed.\n");
        return -1;
    } else if(child == 0) {
        return realMain(argc, argv);
    } else {
        int status;
        waitpid(child, &status, 0);

        if(WIFEXITED(status)){
            // If ok exit, return with status.
            return WEXITSTATUS(status);
        }
        // Pretend like everything is okay, exit quietly.
        return 0;
    }
}
```

**Listing 5:** Wrapper of an application, if the target crashed it will replace the return code with 0.

The second technique we tried was to make use of signal handlers. By catching signals sent to the process, the target can choose what it wants the fuzzer framework to see. The process can ignore the signals and exit cleanly upon reception of crashing signal or do something else entirely. Only signal 9, `SIGKILL` cannot be intercepted by a target application. We tried this method and discovered that some crashes were

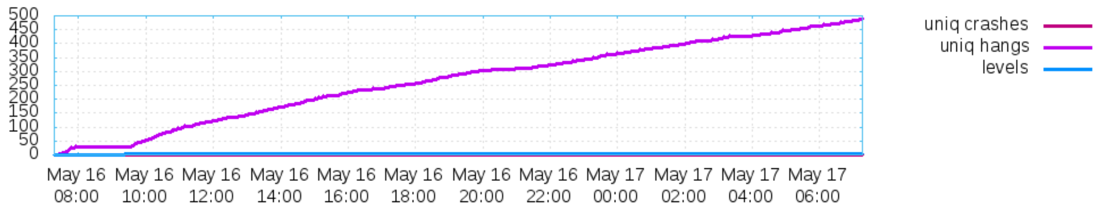


**Figure 5.1:** Graph showing the number of hangs when fuzzing YAN010001 using AFL with the fork technique applied.

still detected by AFL. It turned out that the instrumented code contained a bug that were sometimes triggered by mutated input from AFL. The issue were specific to one of the compilers shipped with AFL and switching to another compiler fixed that issue. Note that these crashing test cases did not crash when used in conjunction with the non instrumented binary. The reason why the signal handler were ignored for the instrumented code is unknown.

The disadvantage of using signal handlers is that it is easy to detect for an adversary if they are looking. There exist tools that can print all system calls that a target application does. This will print the signal handlers in plain text which makes it obvious that further investigation is required by an adversary. Another way is to look in `/proc/<pid>/status`: this file lists all signals that are ignored, blocked or handled by the program with PID `<pid>`.

### 5.3.2 Active Identification

The instrumentation injected into a target application will have a shared memory region where branch counts are updated. In order for the target application to be able to find this shared memory, AFL will communicate its ID number through an environmental variable. This means that a target application can read this environmental variable and then access the shared memory at run-time. This approach can also be used for identifying whether the application is being fuzzed by AFL since it is the responsibility of AFL to setup the shared memory. A proof-of-concept for actively identifying AFL can be seen in Listing 6.

Since the target application has access to the shared memory, it is theoretically possible to actively change the path metrics that AFL uses. The target application can make it seem that more paths has been taken than is actually true, or the reverse.

## 5.4 Honggfuzz

As described in Section 3.3.1 Honggfuzz make use of `ptrace` to observe the target application. This enables Honggfuzz to halt the target directly when signals are sent from the operating system and thus prevents it from hiding signals. The technique

```c
int runningUnderAFL() {
    char env[13];
    // Hack for supporting both QEMU and regular instrumented mode
    snprintf(env, sizeof(env), "%s%s%s", "__AFL", "_SHM", "_ID");
    char *shm_str = getenv(env);
    if(shm_str) {
        int shm_id = atoi(shm_str); // string to int
        void* shmem = shmat(shm_id, NULL, 0);
        // Check if we have correctly attached to a shared memory region
        if(shmem != (void*)-1) {
            printf("Running in afl-fuzz\n");
            return 1;
        }
    }
    return 0;
}
```

**Listing 6:** Proof-of-concept code for active detection of AFL. This code will return `true` if running under AFL, else `false`

where signal handlers are used to mask crashes, as discussed in Section 5.3, will not work when `ptrace` is used. In order to mask crashes, a target observed by `ptrace` must therefore never cause a signal to be generated in the first place.

To counter Honggfuzz, an active approach was used to detect if `ptrace` have attached to the target process. As Listing 7 show, the function `ptrace(PTRACE_TRACEME , ...)` allows a target to determine if a process is being traced by `ptrace`. The system call is actually used by a tracee to tell a tracer to trace the tracee, but since `ptrace` only allows one tracer at any given time, the system call will fail if a process has already been attached and the conditional statement in line 4 in Listing 7 will be true. Note that this technique does not identify Honggfuzz per se, but detects `ptrace`. This means that any other program that make use of `ptrace` will also be fooled, such as `strace`.

```c
1  int realMain();
2  int fakeMain();
3  int main(int argc, char const* argv[]) {
4      if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
5          // Ptrace have attached to the process, do alternative branching
6          return fakeMain();
7      }
8      return realMain();
9  }
```

**Listing 7:** A simple wrapper of a target. Upon detection of `ptrace` Honggfuzz is assumed to be in use and the alternative, `fakeMain()` branch is executed.

The active technique, presented in Listing 7, can be further enhanced by applying the general technique discussed in 5.2. Instead of using `fakeMain()`, you would use `realMain()` but with the input switched to a safe variant. This approach would be advantageous since it would emulate the target application as much as possible without having to worry about it crashing, thus any collected metrics would look legitimate.

## 5.5 Outcome

Two anti-fuzzing algorithms were implemented in the targets, one for AFL and one for Honggfuzz. The targets that crashed during the fuzzing evaluation were re-tested with anti-fuzzing capabilities built-in. For AFL the forking approach was used despite the knowledge that some of the crashes being interpreted as hangs. For Honggfuzz the active approach presented in Section 5.4 were used.

None of the fuzzers were able to detect any crashes in the targets (see Table 5.1).

| Target (with Anti-Fuzzing) | AFL | Honggfuzz |
|---|---|---|
| CADET00001 | ✗ | ✗ |
| EAGLE00005 | ✗ | ✗ |
| NRFIN00010 | ✗ | ✗ |
| YAN0100001 | ✗ | ✗ |
| YAN0100003 | ✗ | ✗ |
| MediaInfo | ✗ | ✗ |

**Table 5.1:** Table showing which of the targets with anti-fuzzing capabilities emitted one or more crashing states for each of the selected fuzzers

If we look at other metrics such as paths explored during a normal fuzzing run, we can see in Figure 5.2 that the number of total paths explored is double that of the paths explored when fuzzing with anti-fuzzing techniques, as seen in Figure 5.3. The number is higher during the normal fuzzing run, but for anti-fuzzing it is not so low as to make an adversary suspicious. We can also see that the pending favourites are the same in both cases. This line denotes the number of favoured entries still waiting to be fuzzed [25].

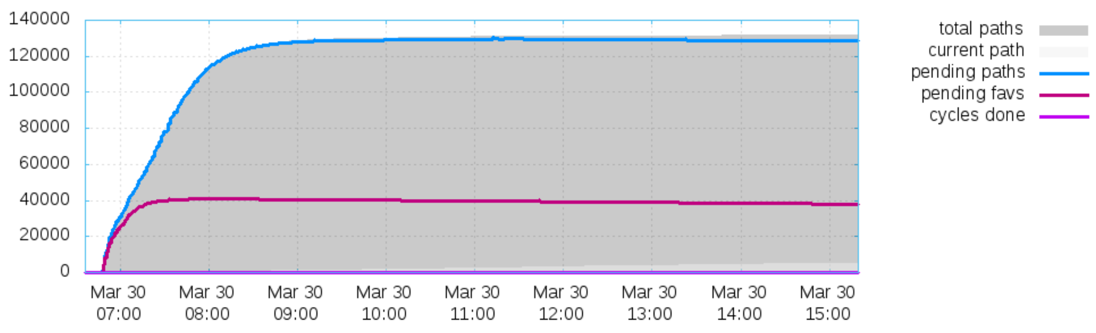


**Figure 5.2:** Graph showing the number of paths explored when fuzzing EAGLE00005 using AFL

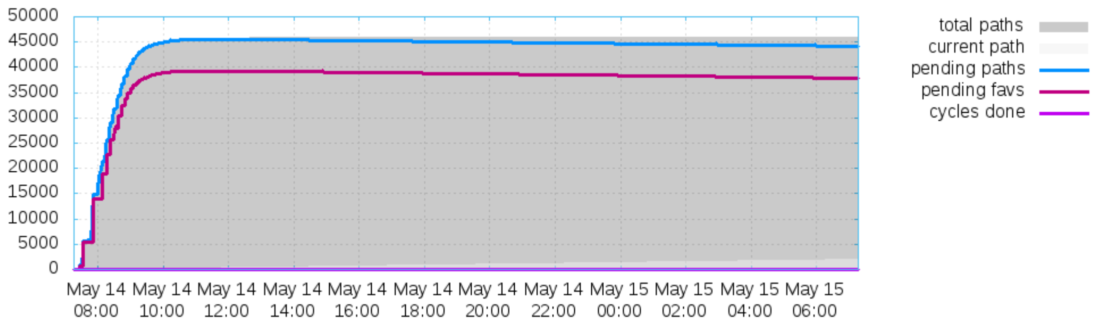


**Figure 5.3:** Graph showing the number of paths explored when fuzzing EAGLE00005 using AFL with anti-fuzzing techniques applied.

# 6
# Ethical considerations

A major part of this thesis work is about deceiving fuzzers. The ability to make it harder for “outside” parties to find bugs in the application that is employing the anti-fuzzing techniques. Applying these techniques will not fix the bugs or vulnerabilities, only make them harder to detect when fuzzing. This could for example be useful for a company that has a closed source application and want to minimise the ability for outsiders to fuzz the application, essentially security by obscurity. This has ethical aspects, since that same company may want to hide malicious code that is discoverable through fuzzing or may not want to employ fuzzing themselves because of time and cost restraints.

An anti-fuzzing technique is not a long-term solution and makes it harder for white-hats, security researchers with good intentions, to find exploitable bugs in software. A malicious adversary may find an exploit through fuzzing, or some other means that good intentioned researchers did not find, thus leaving users vulnerable for a longer time. Therefore one should not rely on anti-fuzzing as a security measurement. However, from a white-hat perspective, it is important to be aware that anti-fuzzing may be employed to mask bugs.

# 7
# Conclusion

In this thesis we did an evaluation of two state-of-the art fuzzers: Honggfuzz and American Fuzzy Lop (AFL). These fuzzers were tested against a real world application, MediaInfo, and a test suite of programs built for evaluating automated vulnerability discovering tools. The results show that AFL was able to outperform Honggfuzz by finding crashes in five target applications in the test suite while Honggfuzz only found crashes in two of the targets when fuzzing for the same amount of time. For MediaInfo, AFL was able to detect 20 unique crashes and Honggfuzz was able to detect two crashes. Even though a vulnerability analysis were not done on the found crashes, the results show that fuzzing is a valuable tool for finding bugs and crashes. Any exploitable crashes most often have to be found manually but fuzzing makes looking for them exceedingly easier.

For the second part of this thesis we set out to research the subject of anti-fuzzing: a concept wherein the target applications applies techniques to prevent a fuzzer from working effectively. These techniques can be used to slow down the whole fuzzing process making it futile fuzzing them, or they can be used for masking or hiding crashes entirely. Our research show that it is possible to implement anti-fuzzing techniques in an application to prevent both AFL and Honggfuzz from finding crashes. Anti-fuzzing can be used to discreetly hide or cover up vulnerabilities in applications and without awareness of anti-fuzzing a fuzz tester may spend a considerable amount of time on a fuzzing process that cannot produce any sane results.

Even though you can do something, does not mean you should. This holds true for anti-fuzzing techniques since the only reason for using it is to either hide maliciously added vulnerabilities or if you suspect there are vulnerabilities and you have not have had enough time to test the software. Anti-fuzzing is security by obscurity since any technique employed is circumventable and thus is only effective at delaying fuzzing. As soon as a technique is discovered and bypassed then new techniques must be invented and a cat and mouse game is created.

## 7.1 Possible improvements

There are possible improvements that can be made to both Honggfuzz and AFL that mitigates the techniques we evaluated in Chapter 5, however, many of these come with negative performance impacts that may be a deal breaker. For example, the fuzzing framework could look at which signals are intercepted by a target application or it could compare which system calls are done when the fuzzer runs the application versus when a user runs it. The latter approach would require manual work by the user and is perhaps not fool proof, but may allow the fuzzer to warn the user of potential anti-fuzzing. However, since anti-fuzzing is not a large problem at the time of writing this, we believe that adoption of such methods will be resisted due to the drawbacks outweighing the potential benefits. One might imagine a fuzzer implementing anti-anti-fuzzing techniques, then performance would probably take a hit and in some sense the anti-fuzzing will have worked due to the fuzzers not being as effective as they could be.

# Bibliography

[1] B. Miller. (2008, Feb.) Foreword for fuzz testing book. [Online]. Available: http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html

[2] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[3] A. K. Patrice Godefroid and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *ACM Sigplan Notices*, vol. 43. ACM, 2008, pp. 206–215.

[4] M. M. Chris Evans and T. Ormandy. (2016, May) Fuzzing at scale. [Online]. Available: https://security.googleblog.com/2011/08/fuzzing-at-scale.html

[5] O. Whitehouse. (2014, Feb.) Introduction to anti-fuzzing: A defence in depth aid. [Online]. Available: https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/

[6] M. Zalewski. (2016, Jan.) american fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[7] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* University of Wisconsin-Madison, Computer Sciences Department, 1995.

[8] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[9] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[10] C. Cadar and K. Sen, “Symbolic execution for software testing: three decades later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” *NDSS ‘16*, 2016.

[12] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, “Automated whitebox fuzz

testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[13] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Security and Privacy (SP), 2015 IEEE Symposium on.* IEEE, 2015, pp. 725–741.

[14] J. Yang, H. Zhang, and J. Fu, "A fuzzing framework based on symbolic execution and combinatorial testing," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing.* IEEE, 2013, pp. 2076–2080.

[15] M. Zalewski. (2016, Jan.) Technical "whitepaper" for afl-fuzz. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt

[16] M. Kerrisk. (2016, Apr.) Linux programmer's manual - signal(7). [Online]. Available: http://man7.org/linux/man-pages/man7/signal.7.html

[17] A. D. Householder, "Well there's your problem: Isolating the crash-inducing bits in a fuzzed file," DTIC Document, Tech. Rep., 2012.

[18] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering.* ACM, 2002, pp. 1–10.

[19] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering.* ACM, 2005, pp. 342–351.

[20] A. Helin. (2016, Apr.) Radamsa. [Online]. Available: https://github.com/aoh/radamsa

[21] Fitblip. (2016, May) Sulley. [Online]. Available: https://github.com/OpenRCE/sulley

[22] Deja vu Security. (2016, Feb.) Peach fuzzer. [Online]. Available: http://community.peachfuzzer.com/

[23] S. Hocevar. (2016, Apr.) zzuf. [Online]. Available: https://github.com/samhocevar/zzuf

[24] Quickfuzz.org. (2016, Feb.) Quickfuzz by cifasis. [Online]. Available: http://quickfuzz.org/

[25] M. Zalewski. (2016, Jan.) Afl readme. [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt

[26] R. Swiecki. (2016, Apr.) Honggfuzz. [Online]. Available: http://http://google.github.io/honggfuzz/

[27] Intel. (2016, Apr.) Intel performance counter monitor - a better way to

measure cpu utilization. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[28] L. T. et al. (2016, Apr.) perf: Linux profiling with performance counters. [Online]. Available: https://perf.wiki.kernel.org

[29] R. Vitillo. (2016, Apr.) Performance tools development. [Online]. Available: http://indico.cern.ch/event/141309/session/4/contribution/20/attachments/126021/178987/RobertoVitillo_FutureTech_EDI.pdf

[30] Darpa. (2016, Mar.) Darpa cyber grand challenge sample challenges. [Online]. Available: https://github.com/CyberGrandChallenge/samples

[31] ——. (2016, Mar.) Darpa cyber grand challenge platform. [Online]. Available: http://www.cybergrandchallenge.com/site/index.html#platform

[32] MediaArea. (2016, May) Mediainfo. [Online]. Available: https://github.com/MediaArea/MediaInfo

[33] M. Zalewski. (2016, May) Fuzzing random programs without execve(). [Online]. Available: https://lcamtuf.blogspot.se/2014/10/fuzzing-binaries-without-execve.html

[34] Mozilla. (2016, Feb.) Fuzzdata. [Online]. Available: https://github.com/MozillaSecurity/fuzzdata

# A

# Appendix 1

## A.1 CGC Seeds

| Target | Seed |
|---|---|
| CADET00001 | <newline> |
| EAGLE00005 | HANGEMHIGH! |
| KPRCA00001 | HELLO<br>AUTH |
| KPRCA00003 | <newline> |
| KPRCA00015 | <newline> |
| NRFIN00003 | <newline> |
| NRFIN00010 | <newline> |
| NRFIN00013 | <newline> |
| TNETS00002 | -1583597902 create cat<br>-1583597903 create hotdog<br>-1583597902 name |
| YAN0100001 | <newline> |
| YAN0100002 | 0<br>0<br>16<br>c<br>c<br>c<br>p |
| YAN0100003 | <newline> |

**Table A.1:** List of the seeds provided to the CGC Challenges

## A.2 MediaInfo seeds

The following is the list of seeds used when fuzzing MediaInfo. The seeds can be found in the fuzzdata repository from Mozilla [34].

- 008b8bb75b8a487dc5aac86c9abb06fb.png
- 194531363df5b73f59c4c0517422f917.jpg
- 1.jp2
- 448636.ogv
- 463696.bmp
- 55abb3cc464305dd554171c3d44cb61f.gif
- audio-quad.wav
- barsandtone.flv
- bear_mpeg4asp_mp3.avi
- black100x100-aspect3to2.webm
- bubblewrap.swf
- bunny_oh2.264
- detodos.mp3
- example4.midi
- favicon.ico
- mpeg4-amr_nb.3gp
- pdf.pdf
- romney.zip
- short-cenc.mp4
- Sorenson.mov
- spacestorm-1000Hz-100ms.ogg
- test27.mp2
- test33.mp1
- webp.webp

## A.3 Hardware specification for test bed

| | |
|---|---|
| **Operating System** | |
| Operating System | Ubuntu 14.04.4 LTS |
| Kernel | Linux 4.2.0-35-generic (x86_64) |
| **CPU 1** | |
| Model | Intel Xeon E5620 |
| Clock speed | 2.4 GHz |
| Cores | 4 |
| Hyper-threading | Yes |
| **CPU 2** | |
| Model | Intel Xeon E5620 |
| Clock speed | 2.4 GHz |
| Cores | 4 |
| Hyper-threading | Yes |
| **RAM** | |
| Memory Type | DDR3 |
| Size | 98304 MB (12 x 8 GB) |
| Speed | 1066 MHz |
| **Hard drive** | |
| Model | Western Digital WD5003AZEX |
| Size | 500GB |
| Rotation Speed | 7200 rpm |

**Table A.2:** Specification of the hardware on the testing platform

## A.4 Asan Triage

The following is a script that compares backtraces of a set of crashing inputs. We used a slightly modified version of this script when fuzzing the CGC targes due to them expecting input through stdin. The script require the targets to be compiled with address sanitizer support for best results. Example invocation: `./asan-triage.py my_folder_with_crashes "./prog -args @@"`

```python
#!/usr/bin/python2

"""
This is free and unencumbered software released into the public domain.
Anyone is free to copy, modify, publish, use, compile, sell, or
distribute this software, either in source code form or as a compiled
binary, for any purpose, commercial or non-commercial, and by any
means.
In jurisdictions that recognize copyright laws, the author or authors
of this software dedicate any and all copyright interest in the
software to the public domain. We make this dedication for the benefit
of the public at large and to the detriment of our heirs and
successors. We intend this dedication to be an overt act of
relinquishment in perpetuity of all present and future rights to this
software under copyright law.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
For more information, please refer to <http://unlicense.org/>
"""

```

```python
import os
import os.path
import argparse
import subprocess
import random

remove_nils = lambda xs: filter(lambda x: x <> '', xs)


if __name__ == "__main__":

    # Arguments
    parser = argparse.ArgumentParser(description='')
    parser.add_argument("seeds", help="", type=str, default=None)
    parser.add_argument("cmd", help="", type=str, default=None)
    parser.add_argument("-d", help="", type=int, default=5)
    parser.add_argument("-p", help="", action="store_true",
↪   default=False)


    options = parser.parse_args()
    seeds = options.seeds
    depth = options.d
    prune = options.p

    cmd = options.cmd
    gdb_cmd = "env -i ASAN_OPTIONS='abort_on_error=1' gdb -batch -ex
↪   'tty /dev/null' -ex run -ex bt 20 --args @@ 2> /dev/null"
    all_files = []
    dedup_files = dict()

    for x, y, files in os.walk(seeds):
        nfiles = len(files)
        for f in files:
            f = f.replace("(","\(")
            f = f.replace(")","\)")
            f = f.replace("$","\$")
            f = f.replace(",","\,")

            all_files.append(x + "/".join(y) + "/" + f)


    random.shuffle(all_files)
    all_files = all_files[:1000]
    nfiles = len(all_files)

    for progress, testcase in enumerate(all_files):
        prepared_cmd = cmd.split("@@")
        prepared_cmd = prepared_cmd[0].split(
            " ") + [testcase] + prepared_cmd[1].split(" ")
        prepared_cmd = remove_nils(prepared_cmd)
        #print prepared_cmd
        out = subprocess.check_output(gdb_cmd.replace(
            "@@", " ".join(prepared_cmd)), shell=True)
        #print out
        backtrace = out.split("#")[1:]
        key = ""
        size = os.path.getsize(testcase)
        dkey = 0
        for x in backtrace:

            if dkey == depth:
                break

            if "??" in x or "__" in x:
                continue
            if " in " in x:
                x = remove_nils(x.split(" "))
                key = key + " " + x[3]
                dkey = dkey + 1

            else:
                x = remove_nils(x.split(" "))
                key = key + " " + x[1]
                dkey = dkey + 1

        # print key
        y = dedup_files.get(key, [])
        dedup_files[key] = y + [(testcase,size)]

    for (k, xs) in dedup_files.items():
        print "*"+k,
        xs = sorted(xs, key=lambda x: x[1])
        for x in xs[:1]:
            print x,

        if prune:
            for x in xs[1:]:
                os.remove(x[0])

        print ""
```