SAPIENZA
UNIVERSITÀ DI ROMA

The Bugs We Left Behind: Evaluating Autonomous Fuzzing Infrastructures for Overlooked Bugs and Vulnerabilities

Facoltà di Ingegneria dell'informazione, informatica e statistica
Cybersecurity

Matteo Sabatini
ID number 1794627

Advisor
Prof. Daniele Cono D'Elia

Co-Advisor
Eng. Matteo Marini

Academic Year 2023/2024

Thesis not yet defended

The Bugs We Left Behind: Evaluating Autonomous Fuzzing Infrastructures for Overlooked Bugs and Vulnerabilities
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: sabatini.1794627@studenti.uniroma1.it

Acknowledgments

I would like to express my gratitude to my thesis advisor, Prof. Daniele Cono D'Elia, for giving me the opportunity to deepen my knowledge in one of my favorites and also most important aspects of cybersecurity. His guidance, support, and encouragement were fundamental in the creation of this work.

I also profoundly thank my thesis co-advisor, Eng. Matteo Marini, for his expertise, suggestions, and patience throughout the development of this thesis.

Abstract

In a world that relies on digital systems, the security of software applications is a crucial property to protect both people and their assets from cyberattacks. To this end, numerous techniques have been proposed that can detect a wide range of vulnerabilities before they can be exploited by malicious entities. The most prominent example of such techniques is *Fuzz Testing*; it works by repeatedly running the software, generating new inputs each time to exercise as many functionalities of the code as possible. The key idea is that by generating different inputs, the *fuzzer* will eventually produce one that is capable of triggering a bug.

Thanks to their *fuzzing strategies*, fuzzers shine in producing inputs that allow them to execute new code as testing time passes. While reaching constantly increasing *coverage*, fuzzers may miss bugs that do not cause observable behaviors at a high level. To tackle this issue, fuzzing is often used in combination with sanitizations, a technique that detects specific silent bugs at run-time. The combination of the two techniques has become the most popular approach to detecting bugs in software.

To ease the deployment of fuzzing, fuzzing frameworks are being proposed. These frameworks are fully autonomous and allow for *continuous testing.* While they usually rely on a private disclosure mechanism for new bugs, these frameworks publicly release lots of data that can be analyzed. With all this accessible data, no bug should be overlooked: an attacker could leverage the available data to find potential vulnerabilities with a much lower effort than running their own fuzzing campaigns, and potentially exploit them to attack the target software.

In this thesis, we are going to evaluate fuzzing frameworks for possibly overlooked bugs. We will be exploring two main directions: first, we are going to evaluate fully autonomous mechanisms, where suboptimal configurations or procedures may cause some bugs to be missed; then, we are going to evaluate another class of frameworks where the human factor is crucial in the detection of new bugs and vulnerabilities. We will show that, unfortunately, these frameworks do indeed miss a relevant number of bugs. These bugs can be detected with little effort using publicly available data, and some of them have security implications. After reporting all the bugs, we identify the fundamental issues that cause these frameworks to miss bugs and propose enhancements to allow them to detect bugs and vulnerabilities more accurately.

Contents

Chapter 1

Introduction

Nowadays, technology has permeated every aspect of human life, allowing everything and everyone to be interconnected anywhere and at any moment. Thanks to its importance, it quickly became one of the most popular fields of work in the modern age, with organizations employing and relying entirely on such technologies to provide their services. However, cybercrime has become very common.

Cybersecurity is an aspect of informatics whose role is ensuring the safety of the technologies available and the people who use them, both in terms of software and infrastructure, protecting systems against security issues and vulnerabilities that could potentially be exploited by malicious entities. In fact, according to the "Consortium for IT Software Quality" (CISQ), poorly designed software cost the economy \$2.08 trillion dollars in 2020 alone [6], numbers that show how security against software bugs and vulnerabilities should be handled with maximum attention.

In this context, *software security* aims to analyze and detect bugs in a program that may pose a security threat to any user and/or organization employing said software. The "National Vulnerability Database" (NVD) classified software vulnerabilities into 5 root causes [46]: input validation, access control, exception handling, configuration errors, and race condition. All these scenarios may lead to a criminal gaining access to your system and executing malicious code, potentially also with the intention of performing privilege escalation and taking ownership of the machine.

One of the 7 components in the "Software Development Life Cycle" (SDLC) is *Testing*, which means analyzing and checking your code to make sure that it satisfies some quality, correctness, and security criteria both in terms of functionalities provided and underlying infrastructure: this may either be done "statically", focusing on prevention and removal of potential defects from source code by means of coding standards and best practices, or "dynamically", analyzing the code during its execution with different inputs.

While testing a project is crucial to ensure its security and provide useful information that can be used to further improve the development of the product, it also involves fixing and solving many different kinds of problems affecting your code, which will obviously require time, resources, knowledge, and effort.

1.1 Context

One of the most important goals of software testing is *finding bugs*, defects in the code that cause unwanted and unexpected results, compromise the security of the product, and may lead to vulnerabilities being exploited by malicious people. In this context, a popular approach is *automated testing*, where developers rely on external software to automatically perform repetitive tests and tasks over long periods of time without any human interaction. An example of an automated testing technique is *fuzzing*, proposed in 1988 [42] and primarily used to automate simple tests previously performed by humans, which gained popularity due to its ability to discover crashes and bugs. Its base idea revolves around feeding (seemingly) random inputs to a program, analyzing its execution flow, and generating new testcases to explore all paths in the code as well as trigger unexpected behavior. This process is repeated across several and extensive fuzzing sessions, while its effectiveness and simplicity of use allow developers from all languages and levels of knowledge to improve the security of their software. It also discovered one of the most famous and critical bugs in history, the “Heartbleed” bug that affected OpenSSL.

Since 2001, with the rise of “Open-Source Software”, ensuring the security of software that was freely available and modifiable by anyone became a top priority: because the source code was freely accessible, it was only a matter of time before cyber-criminals began exploiting their vulnerabilities. Moreover, many modern paid applications often rely on such free software to provide their services.

Given their popularity, Google announced the ClusterFuzz project in 2012, a cloud-based fuzzing infrastructure for testing security-critical components of the Chromium web browser, where fuzzer developers could upload their fuzzing tool and be rewarded if their product detected a crash in the browser. In 2016, this infrastructure was repurposed as the back-end for OSS-Fuzz, a new campaign that provided continuous fuzzing at Google-scale for open-source developers. Later, in 2021, Google announced yet another new project called FuzzBench, which embraced the original purpose of ClusterFuzz and focused on supporting the development of open-source fuzzers thanks to tests based on real-world benchmarks and daily reports. All these frameworks use Google Cloud as their common sharing infrastructure, allowing them to easily share and access data independently. These automated fuzz-testing infrastructures are also valuable to researchers, who can use them to analyze and understand how open-source projects are developed and tested, access the same resources used by their developers during testing, and perform fuzzing locally for educational and research purposes.

While test automation is obviously a time-efficient solution compared to manual assessment, allowing machines to perfectly replicate the same repetitive task continuously and over extended periods of time with high consistency and accuracy, configuration and design choices may result in a partial loss of information. For this reason, the aforementioned automated fuzz-testing infrastructures might be missing bugs during their testing, and that is the main focus of this work.

1.2 Thesis Idea and Contributions

This thesis focused on analyzing the workflow and results produced by automated testing infrastructures, more specifically OSS-Fuzz and FuzzBench: although these campaigns have enormous computing capabilities and have been performing continuous fuzzing for several years, they still suffer from machine errors and limitations due to their design choices. This work will show that these frameworks are in fact missing a relevant number of bugs, discovered by rebuilding a subset of the programs that are part of these campaigns and testing them locally using a set of novel tools and techniques for security analysis. All bugs found were then manually assessed, debugged (when possible), and finally reported to the respective developers hoping that this would help them make their software more secure.

Contributions. To summarize, this thesis makes the following contributions:

- A study of the accuracy of existing continuous fuzzing frameworks.
- The detection of sources of overlooked bugs on state-of-the-art fuzzing frameworks.
- An assessment of the relevance of software sanitizers in fuzzing campaigns.
- The responsible disclosure of almost 250 bugs to the respective project maintainers.

1.3 Outline

This thesis is structured as follows. Chapter 2 provides all the necessary concepts to understand what is fuzzing and how a fuzzer works, along with notions about some tools that are widely used in this field. It also introduces the definition of “Open-Source Program” and the automated testing infrastructures analyzed in this work. Chapter 3 discusses the methodology applied during the selection of the projects and the test phase, including the problems faced during this process and the solutions found. Chapter 4 shows the results obtained, analyzing and talking about what could be inferred from them, and discusses the reports issued and the developers’ responses. Chapter 5 ends this thesis by summarizing its contents, presenting some final considerations about fuzz testing, and introducing possible future works to expand this line of research.

Chapter 2

Background

This chapter will provide all the necessary background information and techniques employed in this thesis. We start by explaining the concept of *fuzz testing*, with an in-depth analysis of how it works, the main components of a fuzzing session, and the different approaches available. Follows an introduction on software sanitizers, highlighting their relevance in the context of fuzzing. Then, we provide a short definition of “Open-Source Software”, the main challenges faced during its development, and its connection with fuzzing in the modern era. Concludes a thorough description of the modern automated testing infrastructures analyzed by this work, detailing their inner working and shortcomings.

2.1 Fuzz Testing

One of the most popular approaches to testing a program is *automated testing*, where developers rely on scripts and specifications passed to external software or a created toolkit that automatically performs tests over long periods of time without human supervision. It can be divided into *static testing*, which is based on a thorough analysis of the source code and uses techniques like symbolic execution to abstract the program execution, and *dynamic testing*, which aims to execute the program many times with the objective of traversing all possible execution paths. It is particularly useful to formalize repetitive tasks into automated operations with high scalability, accuracy, consistency, and fast execution times. Additionally, programmers may prepare and use such pre-programmed tools with minimal costs after an initial effort in developing them, making it a crucial integration for large codebases and making sure that the product meets quality standards before being released. However, manual evaluation is still required and irreplaceable, both to ensure the correctness of the operations performed as well as to maintain and keep these tools effective and up to date with the software development [58].

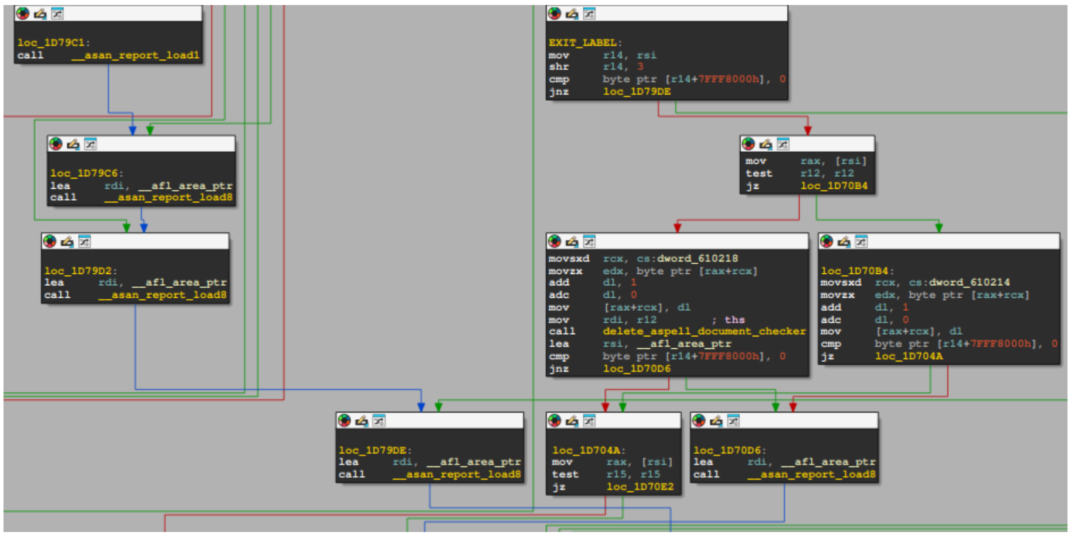
Among dynamic automated testing techniques, *fuzz testing* (or *fuzzing*) has become increasingly popular and widespread due to its ability to trigger critical issues such as crashes, assertions, memory leaks and undefined behavior, done by systematically testing the program using random, malformed and unexpected inputs. This allows developers to test their programs for so-called “corner-cases”, i.e. situations that are difficult or complex to reproduce when the program is being properly used, but that could lead to unexpected and/or unwanted behavior potentially exploitable by malicious people and therefore should be handled properly. It also provides great flexibility, as this technique can be applied to an entire binary or selected small portions of code, introducing specific testing functions.

We define 3 types of fuzz testing:

- **application fuzzing:** used for UI elements (such as buttons, input fields) or command-line programs, tests may include high-frequency inputs, providing random/invalid content and inputs exceeding the expected size
- **protocol fuzzing:** used to test the behavior of network elements and servers when invalid messages are sent over a chosen protocol, useful to ensure that such content is not misinterpreted and potentially executed as commands
- **file format fuzzing:** used for programs that accept "structured inputs", i.e. files that have a precise and standard format (like .doc, .jpg), whose structure and content are altered to trigger unwanted behavior

The main concept of fuzzing revolves around testing a program on many (seemingly) random inputs over several *fuzzing sessions*, each time discovering a bit more about the structure of the program, and then using this knowledge to create new testcases that attempt to trigger previously unseen execution paths and bugs: when unexpected behavior is observed (i.e. crash, assertion, bugs, ...), the related input is reported to the user and saved for bug analysis, described in the following paragraphs. Each fuzzing session usually takes several hours or days to produce meaningful results; however, because this technique is mostly based on randomness, it also means that results may vastly differ between sessions, even across tests using the same configuration and resources. For this reason, multiple sessions are necessary before a conclusion regarding the safety of the program can be reached.

One of the most popular techniques to determine whether a particular test case was useful or not towards the final results is to analyze *code coverage*: this metric represents a percentage that defines how much of the source code has been executed during a single fuzzing session, with the assumption that a high value means that most of the code has already been explored and there is a lower chance of having undetected bugs coming from already tested execution flows.



```
loc_1D79C1:
call     __asan_report_load1
```

```
loc_1D79C6:
lea      rdi, __afl_area_ptr
call     __asan_report_load8
```

```
loc_1D79D2:
lea      rdi, __afl_area_ptr
call     __asan_report_load8
```

```
EXIT_LABEL:
mov      r14, rsi
shr      r14, 3
cmp      byte ptr [r14+7FFF8000h], 0
jnz      loc_1D79DE
```

```
mov      rax, [rsi]
test     r12, r12
jz       loc_1D70B4
```

```
movsxd   rcx, cs:dword_610218
movzx    edx, byte ptr [rax+rcx]
add      dl, 1
adc      dl, 0
mov      [rax+rcx], dl
mov      rdi, r12          ; ths
call     delete_aspell_document_checker
lea      rsi, __afl_area_ptr
cmp      byte ptr [r14+7FFF8000h], 0
jnz      loc_1D70D6
```

```
loc_1D70B4:
movsxd   rcx, cs:dword_610214
movzx    edx, byte ptr [rax+rcx]
add      dl, 1
adc      dl, 0
mov      [rax+rcx], dl
cmp      byte ptr [r14+7FFF8000h], 0
jz       loc_1D704A
```

```
loc_1D79DE:
lea      rdi, __afl_area_ptr
call     __asan_report_load8
```

```
loc_1D704A:
mov      rax, [rsi]
test     r15, r15
jz       loc_1D70E2
```

```
loc_1D70D6:
lea      rdi, __afl_area_ptr
call     __asan_report_load8
```

Figure 2.1. CGF taken from a sample binary provided as input to IDA Disassembler [26]

This is usually done by creating a “Control Flow Graph” (CFG) of the program, a representation in graph notation of all paths available during a single execution flow: each node in the graph represents a “basic block”, i.e. a sequence of instructions or statements with no jumps, and the edges between nodes represent jumps, loops and conditional branches in the program’s execution (see Figure 2.1). To determine the current coverage achieved, each execution flow is analyzed and the following metrics are measured: number of nodes and edges traversed, number of functions triggered, and conditional branches activated. All this information allows the fuzzer to keep track of all paths that have yet to be explored. Given this, we define *interesting input* any testcase that increases the code coverage achieved in a fuzzing session, and these will be as a reference when generating new testcases to hopefully get even more coverage.

2.1.1 What Is a Fuzzer

A *fuzzer* is the tool performing fuzz testing, taking as inputs the program being tested and a set of testcases that will be used during the fuzzing session. It is responsible for executing the program against all the provided inputs in an automated manner, analyzing each execution for potential unwanted behavior, and producing new and useful testcases for future fuzzing sessions that will attempt to trigger new execution flows and even more bugs.

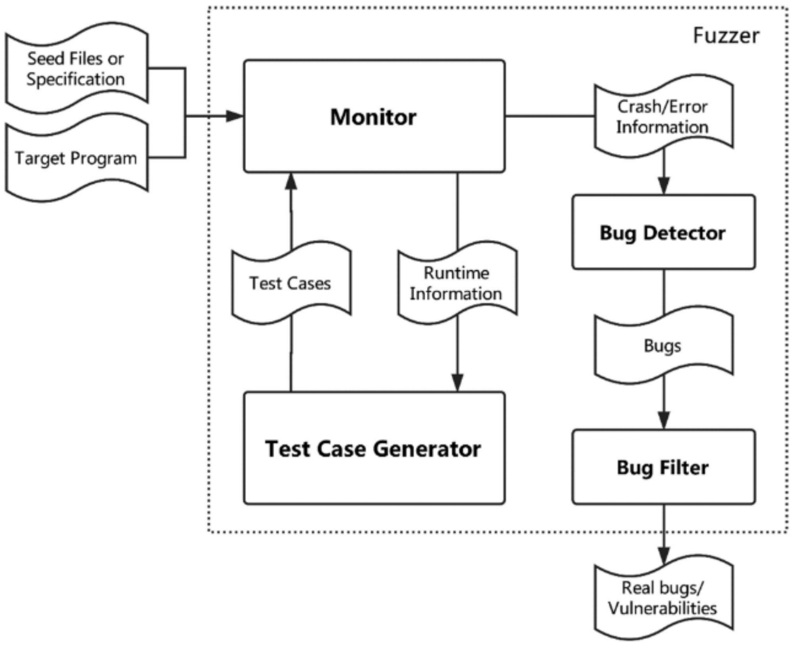


Figure 2.2. General fuzzing workflow [33]

In general, a fuzzer is composed of the following key elements [1] [33].

Target program. Refers to the program that is being evaluated: it could be either binary or source code, although source code of real-world software is usually not accessible.

Program's specification. Some fuzzers can receive as input the program's specification, a formal description of the target program using as reference the available documentation and source code, which translates its functionalities and expected behavior using a specific grammar that the fuzzer understands. However, this information is rarely available and often too difficult to infer without the appropriate data.

Observer (or Monitor). Provides runtime information observed during the execution of the target to the fuzzer, leveraging techniques such as code instrumentation, taint analysis, code coverage, and many more. Such information may be relatively simple, like the total running time for a test and its output, to more advanced ones, like the maximum depth of the stack. They are usually not preserved across executions unless an "interesting input" is encountered, in which case they are relayed to the *Test case generator* to be used as reference during the generation of new testcases.

Executor. Responsible for defining how the program will be executed and the arguments passed on each run. The input for a single test is provided either by writing it in some specific memory location or passed as argument to a so-called "harness function", although each fuzzer has its own implementation of this element. Given this, we briefly mention a few standard functionalities that compose this element. The *InProcessExecutor* runs the "harness function" inside the fuzzer and provides crash detection. The *ForkServerExecutor* is responsible for spawning and fuzzing the different child processes created by the tested program. The *TimeoutExecutor* wraps and installs a timeout for another running executor when the executed inputs takes too long.

Test case generator. Performs generation of new testcases to increase code coverage using runtime information received from the observer and *interesting inputs* as reference. This can be done using either *mutation* or *generation*, depending on the type of fuzzer chosen (discussed later).

Feedback. Chain of components that classify the result of a single execution and determine if the initial input is "interesting" or not analyzing the information received from the observers and the updated coverage map. Some examples shown in Figure 2.2 are the *Bug Detector*, which collects and analyzes debug information during crashes and errors, and the *Bug Filter*, which classifies the errors collected. Each component of this chain usually has its own objective (crashes, timeouts, new execution flows discovered, assertions), and combining them in a boolean expression allows the developer to collect more fine-grained results and maximize a desired target function.

Testcase. Data taken from an external source and provided as input to the tested program to observe its behavior. Each testcase is defined by a set of metadata like ID, description, and expected result. The first fuzzing session takes a set of inputs defined and provided by the developer itself, while future fuzzing sessions will also rely on previously discovered "interesting inputs".

Corpus. Location where testcases are stored, usually disk or memory. An *input corpus* may be composed of several testcases with the same properties, like crashing the program under a specific situation. An *output corpus* may be composed of all testcases that are considered "interesting" or a collection of all inputs that triggered a bug in the program.

Fuzzers can then be categorized according to the 3 following characteristics.

Input generation. During each fuzzing session, one of the most important operations performed by the fuzzer is the generation of new testcases, with the objective of diversifying and increasing the effectiveness of the corpora that will be used on future fuzzing sessions: more specifically, an effective fuzzer should be capable of generating inputs "valid enough" so that they are not rejected by the program's parsers, but also "invalid enough" to potentially trigger corner-cases. Given this, we distinguish between *mutation-based* and *generation-based* fuzzers. The *mutation-based fuzzers* require an initial seed of inputs as reference, and the generation of new inputs is performed by applying "mutators" on the provided seeds: these operations include flipping/adding/removing bits (or bytes), performing mathematical operations and sometimes even completely randomizing their content. Finally, these mutations are usually mixed forming sequences to increase randomness. The *generation-based fuzzers*, instead, rely on a good source of randomness to generate new inputs from scratch, and for this reason they do not depend on the existence of a good initial corpus nor its quality, although they require more starting time before the tests become effective.

Input's structure awareness. Another information useful to the fuzzer, although not always available, is the "input model", i.e. the correct structure that a file must have when testing programs that follow rigorous standards. However, not all of them provide this feature, and for this reason we define *smart* and *dumb* fuzzers. A *smart mutation fuzzer* might leverage this knowledge to switch between different types of inputs, while a *dumb mutation fuzzer* can only rely on the structure of the "interesting inputs" and apply limited random mutations, usually resulting in a much lower proportion of valid inputs being generated. A *smart generation fuzzer* uses this information to avoid wasting resources on generating inputs that are too random to be accepted, as the *dumb generation fuzzer* attempts to generate new inputs without any reference, oftentimes putting more stress on the program's parsers rather than the program itself due to the overwhelmingly high number of invalid inputs generated in the first phases.

Program's structure awareness. Fuzzers may be accompanied by program analysis techniques to increase the efficiency and effectiveness of the tests performed. A *black-box fuzzer* is completely unaware of the program's structure, therefore assumes the program as a simple machine that takes a random input and generates an according output: this approach is relatively fast, can be easily parallelized and has good scalability, however it will most likely find only bugs that do not require particular conditions to be triggered, also called "surface bugs". A *white-box fuzzer* employs program analysis techniques to systematically explore and reach critical program locations through meticulously crafted inputs, allowing you to discover bugs that could be potentially hidden deep in the program: while this approach is arguably the most effective one, it implies that bugs related to unknown aspects of the program can be easily missed, and the time used to analyze the program as well as generating such specialized input exponentially increases with the program's complexity. A *gray-box fuzzer* attempts to find a balance between efficiency and effectiveness by integrating the best aspects of both approaches: using a minimal amount of knowledge of the program's structure to achieve a sufficient degree of code coverage such that the results obtained are satisfactory.

A *fuzzing session* defines the process of testing a program using a fuzzing tool. Assuming the program has been properly compiled using the chosen fuzzer, the first step is to simply start the fuzzing session and let the fuzzer execute the program on many different testcases. After some time, when either the initial corpus has been exhausted (mutation-based) or the fuzzer learned what is an acceptable input (generation-based), it starts applying random operations to generate new testcases.

During each execution, code coverage and unexpected behaviors are tracked, so that all testcases that increase any of these statistics will be regarded as "interesting inputs" and saved in a separate queue. In this context, the fuzzer has to be sensible enough to distinguish between crashing and non-crashing inputs without having full knowledge over the program tested, and therefore *sanitizers* are used to "instrument" the source code and inject assertions that make the program crash when a particular kind of failure is detected. Section 2.2 explains this concept more thoroughly.

At the end of each fuzzing session, the user is presented with three results: a collection of statistics regarding the coverage achieved and bugs found, a list of inputs that caused crashes along with some metadata (testcase ID, bug type, memory state, etc...), and a set of "interesting inputs" that can be used in future fuzzing sessions to provide the fuzzer with even more information about the structure of a good input that explored the deepest parts of the program. The statistics provided may be useful to understand how effective this particular fuzzing session was with respect to the previous ones, if the initial corpus needs to be refined, and whether the changes applied to the code have been fruitful or not. The new set of "interesting inputs" is then added to the existing initial corpus, which in turn is *pruned*: this process removes any duplicates and inputs that trigger the same execution flow or bugs, which is crucial to ensure that the size of the corpus does not explode over time. Finally, the developer has to analyze all the inputs that caused a bug and perform *bug triage*: execute each input individually to observe its output, determine which kind of error occurred and why, fix the bug entirely (if possible) or at least patch the problem, and ensure that the bug does not occur in future fuzzing sessions by including the triggering input(s) in the set used for the tests.

All these steps are repeated across many different fuzzing sessions, sometimes even changing the scope of the tests to increase code coverage or stressing the program with buggy inputs, each time making the program more secure against vulnerabilities and more robust to extraneous inputs.

In conclusion, many modern fuzzers originate from a solution that represented a breakthrough in fuzzing research called *American Fuzzy Lop* (also known as *AFL* [62]), a "dumb mutation-based fuzzer" that achieved many results over the years: in September 2014 it discovered "Shellshock" [4] (also known as "Bashdoor"), a family of security bugs affecting the Unix Bash shell that allowed malicious users to execute arbitrary commands without confirmation, while in April 2015 it discovered the famous "Heartbleed" [25] bug in OpenSSL, which allowed malicious users to decipher the otherwise encrypted communications used by the TLS protocol. Since 2017, when the development of this tool was stopped, there have been many variants and improvements of this tool, and some honorable mentions are WinAFL [63], VUzzer [50], FairFuzz [32], Fuzzolic [3] and QSYM [61]. This thesis focused on AFL++ [5], a direct fork of AFL with new and better mutation and coverage algorithms, which is the current state-of-the-art fuzzer.

2.2 Sanitizers

A *code sanitizer* is a tool used to detect bugs in a program during both compilation and runtime, with different sanitizers specializing in detecting various kinds of bugs: misuse of addresses, stack-based and heap-based overflows, use of uninitialized memory, memory leaks, memory corruption, and undefined behavior.

They are added to a program through "instrumentation", a technique that refers to modifying either the source code or the binary code to introduce some additional functionalities and references that can be used by other tools to perform code analysis, logging, and profiling. Although code sanitizers should be a standard practice in the development cycle of a program, introducing these tools requires knowledge and extensive tests to check for potential errors, also because they do not always interact very well with shared libraries or other external dependencies. It is also extremely important to mention that these bug detection tools are not meant to be linked against production executables, as their runtime was not developed with security-sensitive constraints and may compromise the security of the released product [35][37][38].

A *compile sanitizer* is a tool that performs instrumentation at compile-time, meaning that it introduces functions and libraries that will be then added and compiled together with the original source code, effectively altering its default behavior. Their advantages are twofold: they can achieve full coverage and provide warnings and errors during the compilation process, since the analysis is performed directly on the source code and includes all possible paths, and they are also able to detect errors during the execution of the program, although limited by the single execution flow analyzed. However, these sanitizers introduce a non-trivial overhead in terms of increased compile time, execution time, and memory usage, which may cause problems when it comes to managing computing resources.

A *binary sanitizer* is a "Dynamic Binary Instrumentation" (DBI) framework that performs "Just-in-Time" (JIT) compilation to introduce additional instructions and analysis callbacks during the execution of a program without modifying the original code, and they often rely on "Dynamic Binary Analysis" (DBA) tools to analyze programs at run-time at the level of the machine code. Also, these sanitizers are dependent on the input given and the resulting execution flow analyzed, therefore they are not capable of achieving full code coverage. Moreover, this analysis requires attaching another external process to instrument the original one, which usually causes massive slowdowns and sometimes might even break the program entirely.

This work focused on using several compile sanitizers developed by Google [18] as part of its tool suite provided to open-source developers, along with a popular binary sanitizer [56] to perform a cross-check on the results produced by the compile sanitizers, as these tools have been proven to be particularly effective when combined with fuzzing due to their ability to trigger bugs.

2.2.1 ASan and LSan

The *Address Sanitizer* [52] is a memory error detection tool for C/C++ that helps developers to find and fix any out-of-bounds accesses to heap, stack and global objects, use-after-free bugs and provides protection against stack-based and heap-based overflow, making it one of the most popular and effective sanitizers.

When allocating bytes for a buffer and performing an access beyond such boundaries, a program will usually crash with an "out-of-bounds exception", but it could also happen that it is valid to access information in the unallocated address, in which case the program will crash unexpectedly and make it difficult for the developer to understand the problem. Moreover, if such addresses are not properly freed, they could leave a so-called "dangling pointer", and thinking that this memory location still holds valid data (maybe even confidential one) implies serious security issues. If an attacker discovers a program with such vulnerabilities, he could use them to crash the application, corrupt or retrieve sensitive data, or even perform a "remote code execution" attack by inserting some malicious code via this buffer overrun and causing the program to jump and execute that particular memory location, creating an attack vector where anything is possible.

To prevent this, the sanitizer uses a shadow memory to map the memory regions allocated by the application and record whether each byte of such areas can be safely accessed by `load` and `store` operations. Each memory region is assigned a shadow counterpart, containing metadata about its size and the offset range that can be used to safely access it, while any attempt to read/write beyond such boundaries will trigger a sanitizer error, as shown below:

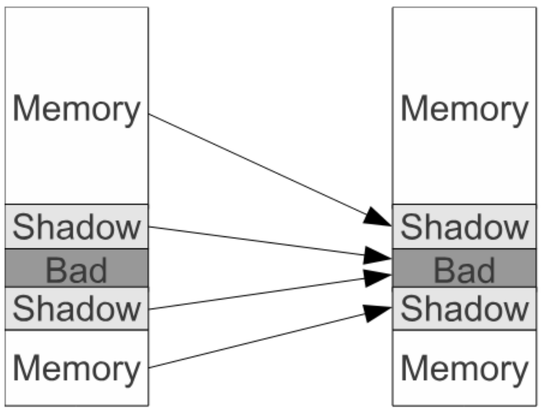


Figure 2.3. Memory mapping of ASan [52]

Detection of out-of-bounds accesses to global variables and stack objects is done in a similar way, i.e. by creating poisoned memory regions around such objects: *global variables* are poisoned at compile-time and their addresses computed during the application startup, while *stack objects* are poisoned and recorded at run-time.

The management of the shadow memory is done by ASan run-time library, containing a specialized implementation of the `malloc` and `free` functions which allocate extra memory for shadow and poisoned memory zones as well as keeping a FIFO stack of allocated and freed memory regions to detect use-after-free, double-free and invalid-free bugs.

This sanitizer is also provided with another component called *Leak Sanitizer* (or *LSan*), a memory leak detector enabled by default that returns which portions of the program are leaking memory as well as the size leaked, thanks to comprehensive and exhaustive stack traces. A memory leak is a condition where a program fails to release memory that is no longer needed due to developers' negligence or software error, effectively reducing the amount of memory available by the machine, and this will inevitably lead to performance degradation (trashing). Although this component may seem very helpful, it usually generates huge logs especially when having complex programs that heavily rely on external libraries, which oftentimes do not perform clean releases of the objects used.

Finally, ASan adds a slight overhead, increasing execution times by an average of 170% and memory usage by 3.4x [52].

2.2.2 MSan

The *Memory Sanitizer* [54] is a memory reads detector for C/C++ that helps developers to find and fix use-of-uninitialized-memory (UUM) bugs, which are considered to be quite tricky as they do not necessarily occur in every execution and could be triggered by any operation performed by the program. The C/C++ languages leave memory management to the developer, who is responsible for correctly allocating, using, and freeing every memory region that is accessed by the program, and for this reason they are also defined to be "memory-unsafe": in fact, unless specified, any new allocation operation performed by these languages creates uninitialized stack and heap objects. Such vulnerabilities may not only be exploited to alter the execution flow and inject malicious code, but also to leak information about a program's internal state, such as the content of the stack or heap.

A UUM bug usually originates from any operation that loads a value from an uninitialized memory region, resulting in an *undefined value* being returned, and operations like conditional branch, syscalls, and pointer dereference are most likely to trigger them. This sanitizer works by using a shadow memory to map each bit of memory used by the program and encode its state (0 initialized, 1 otherwise): all newly allocated memory is poisoned, i.e. the corresponding shadow memory is initialized to `0xFF`, and "shadow propagation" operations are performed to safely copy an uninitialized variable to different memory regions as well as safely perform simple logic and arithmetic operations on them without occurring in errors.

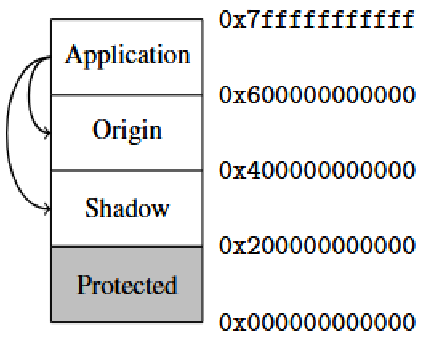


Figure 2.4. Memory mapping of MSan [54]

Given that UUM bugs are notoriously hard to reproduce and debug, MSan provides the “origin-tracking-mode” to obtain more comprehensive and descriptive stack-traces. If necessary, there is also the “advanced-origin-tracking-mode”, which records the entire stack and all operations that happen between the allocation and load of uninitialized variables, but its usability is still discussed due to the huge amounts of memory it requires.

The management of the shadow memory is done by MSan run-time library, which maps the “shadow” and (optional) “origin” areas and marks as uninitialized any new allocated regions as well as the deallocated ones. To update the shadow region, a large subset of the standard `libc` functions are intercepted.

Fixing these bugs generally requires little to no effort, but many developers avoid this sanitizer due to the huge slowdown that it introduces: MSan increases execution times by an average of 300% and memory usage by 2x on short programs, values that rapidly worsen with complex programs [54].

2.2.3 UBSan

The *Undefined Behavior Sanitizer* [38] is a fast undefined-behavior detector for C/C++ that helps developers to find and fix undefined-behavior (UB) bugs. The C language specification allows developers to perform many complex and (sometimes) unreasonable operations on the assumption that they know what they are doing, and this leads to the compiler making arbitrary decisions when the code does not conform to some expected values: for example, an array indexed with an out-of-bound value will not always crash the program if said memory location contains valid content that belongs to the program, while performing a conversion between two data types of different sizes will result in unpredictable loss of data due to rounding operations. This means that different compilers will handle such situations differently and produce unexpected results each time they are executed, all without taking into account that the hardware used and the level of optimization chosen will also affect these results.

Common operations that may lead to undefined behaviors are the following: array subscription out of bounds, overflows/underflows of numerical types due to mathematical or logical operations, shifts of numerical values, dereferenced, misaligned or null pointers, and invalid conversions between data types. Although this list is not exhaustive of all possible scenarios, it is apparent that most of the problems mentioned could be fixed with negligible effort and avoided altogether by applying appropriate coding standards.

This sanitizer works similarly and sometimes overlaps with ASan, putting protected regions around buffers and inserting correctness-checking functions after all mathematical, logical, and implicit/explicit cast operations. By default, due to its simplistic nature, the tool does not print stack-traces and uses a minimal run-time library, although the “print-stacktrace-mode” option can be enabled to obtain more comprehensive results.

To conclude, UBSan adds a trivial overhead, increasing execution times by an average of 120% and memory usage by 2x [38].

2.2.4 Valgrind

Valgrind is a “Dynamic Binary Instrumentation” (DBI) framework that allows developers to build dynamic analysis tools, i.e. tools that perform code analysis at runtime and capable of achieving full coverage of user-mode code. It is distributed with seven production-quality tools that perform code profiling, memory analysis, and threading bugs, with similar functionalities to ASan and MSan [56].

The tool relies on dynamic binary recompilation to load the client program into the “Valgrind Core” and (re)compile the client’s machine code in small blocks using a “Just-In-Time” (JIT) approach, producing a so-called “Intermediate Representation” (IR) which is then instrumented with analysis code. This translated code is then stored in a code cache, so that it can be rerun when necessary avoiding this overhead. It is also capable of monitoring the CPU registers during the program execution by taking control of the real CPU and (conceptually) running the client program on a simulated one, while techniques similar to “shadow memory” are used to analyze and protect both memory and CPU registers [45] [44].

This work relied on Valgrind’s memory-analysis tool called *Memcheck* [55][53], a memory-error detector for C/C++ program when performing accesses to heap and stack elements, use of uninitialized and undefined values, incorrect allocation and freeing of heap memory, misuse of allocation functions and finally tracking memory leaks. Since most of its functions overlap with MSan, it was used as a replacement for detecting UUM bugs when analyzing projects that relied heavily on external libraries: this is because, of the compile sanitizers mentioned above, MSan is the only one that requires all libraries linked to the executable to be rebuilt with this sanitizer to get meaningful results. However, this process proved to be too cumbersome and beyond the scope of this work, also because most projects used statically linked libraries rather than compiling them from scratch.

While Valgrind is a very popular and widely used tool, it also suffers from a major limitation due to its design choice, which is that it is a single-threaded program: this means that its execution cannot be parallelized and the entire program and its analysis run on a single CPU core and kernel thread. In fact, Valgrind applies an average slowdown of 4x by default, value that can range from 10x up to 100x for CPU-bound programs when other analysis tools are introduced [45]: for example, the Memcheck tool introduces a slowdown between 20x-30x for most programs [55].

2.3 Open-Source Software

The *Open-Source Software* is a computer software developed in a collaborative and public manner, released under a particular license that allows other users to freely use, study, modify, and distribute the software and its source code for any purpose: this allows many users to actively participate in the development of a software by proposing changes and new improvements. To be eligible as open-source software, the license's distribution terms must comply with several criteria [47]: the main concept is to let anyone easily download your program and access its source code, allowing them also to modify and redistribute the project without additional fees as long as the original code is appropriately credited.

Given all this, one could argue that making all this information publicly available poses a real threat to security: history has shown us many times that, given enough time and resources, releasing the source code of a program will result in malicious users discovering bugs and vulnerabilities that could have potentially catastrophic consequences. For this reason, we mention some key aspects that should be considered when approaching open-source software.

Development. Open-source software is usually released under two development branches: a "stable" version, composed of all the functionalities that have been thoroughly tested and work as intended, and a "build" version, that is slightly buggier as it includes proposed changes and new features that have yet to be refined. Releasing the "build" version early not only allows the developers to showcase their work and attract even people, but also provides them with feedback from real users that are willing to run untested versions of their software.

User interaction. Providing full access to the source code means that other users might want to contribute to the program's development and help the developers in improving and refining the product: considering that each user may have different knowledge and programming skills as well as different testing environments, this allows them to test and benchmark the product on a wide range of systems, further increasing the probabilities of finding new and unknown bugs that may be specific to a single OS or architecture.

Bug reporting. Although any user has the right to mention a bug, error, or mistake in the program or the documentation, it is still up to the developers to ensure the truthfulness of what has been reported and how to tackle it. For example, bugs that are not security-relevant or that may be related to "Quality of Life" (QoL) aspects are easily pushed back as secondary problems or simply ignored altogether. Sometimes, if the developers are kind enough to accept your request but do not have time and resources to solve it, they might ask for a proposed fix and cite the user in the next patch notes as a way of thanking them.

2.4 Continuous Fuzzing

In the past decade, fuzzing became one of the most popular dynamic automated testing techniques, with both small and large companies employing it to discover bugs and vulnerabilities in their products. However, a fuzzing session should run for at least 24 hours [29] to produce meaningful results, and time is a resource that most organizations cannot afford to lose. For this reason, they often employ short but continuous fuzzing campaigns together with a software development paradigm called *CI/CD pipeline* (*Continuous Integration* and *Continuous Delivery*): CI refers to the practice of frequently integrating and testing changes to the source code, to ensure that the product is always in a well-functioning state. CD is a delivery strategy in which software is developed in short tasks and released with incremental updates, helping developers to reduce costs by making development simple and repeatable, but also avoiding unnecessary risks from applying too many changes at once.

In general, a continuous fuzzing framework refers to an automated framework that integrates any changes committed to the code, builds and tests debug versions of the program with one (or more) configuration of fuzzers and sanitizers, and finally deploys a new version of the product to end users. To further increase early defect discovery, many of these frameworks rely on a technique called *ensemble fuzzing*, with a workflow similar to the one shown below:

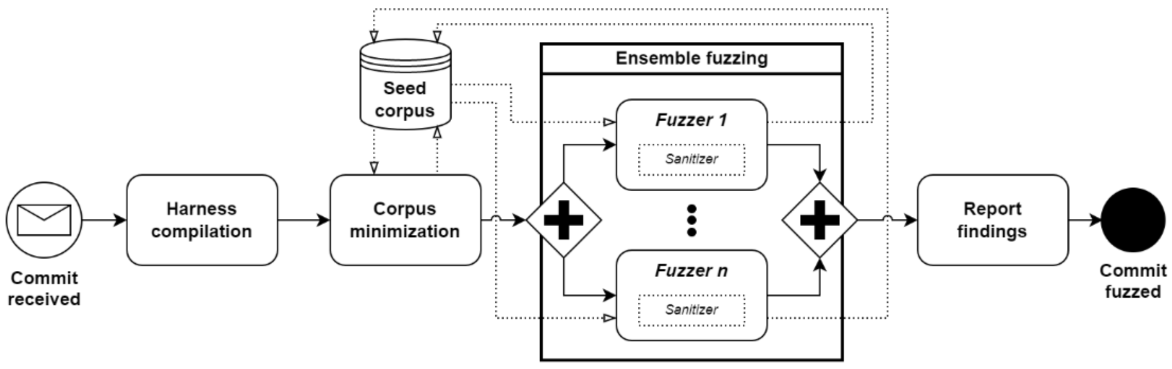


Figure 2.5. Generic design of ensemble fuzzing in a CI/CD pipeline [30]

Instead of focusing on testing a program with a single fuzzer (along with sanitizers) for several hours, this approach focuses on testing the same program in short sessions with different combinations of fuzzers and sanitizers, mixing and matching mutation-based and generation-based fuzzing tools. Each fuzzer runs asynchronously and separately from the others, collecting interesting inputs and applying its own operations to generate new inputs. Then, all results are collected and synchronized into a single queue for analysis and use in subsequent fuzzing sessions: this approach maximizes both the diversity of inputs used, as each fuzzer will be using testcases produced by other fuzzers, and the diversity of results, as each fuzzer will test code differently potentially leading to the discovery of new bugs. Of course, in order to use this technique, developers must compile and prepare the binaries accordingly.

We briefly mention a few popular continuous fuzzing infrastructures. Microsoft announced in 2020 the *Project OneFuzz Framework* [40], an extensible and open-source self-hosted fuzz testing framework with the goal of enabling developers to easily and continuously fuzz programs prior to their release while scaling fuzzing workloads in the cloud thanks to Azure. It allowed developers to build and compose a continuous fuzzing workflow using several fuzzing tools, perform automatic bug triage and results deduplication. Unfortunately, its development was stopped as of September 2023 [41]. GitLab provides continuous coverage-guided fuzz testing as part of its CI/CD pipeline integration [8], supporting many different languages and fuzzing engines. Google provides *ClusterFuzzLite* [12], a continuous fuzzing solution as part of CI workflows to fuzz pull requests and catch bugs before they are committed [11], and *OSS-Fuzz* [51], a continuous fuzzing service for open-source software.

Google was also one of the first major companies that connected automated testing, continuous fuzzing frameworks and open-source software together. The *Google Open Source Project* [17] is a campaign started in 2004, one of the oldest open-source campaigns in the industry: it was initially meant to share Google-developed software under open licenses, with the intention of bringing free technology and information sharing to the public, but it quickly became a program dedicated to improving the open-source software ecosystem as a whole. Thanks to this campaign, many projects became popular and gained worldwide recognition: Android OS, TensorFlow, the Go programming language, and many more.

This thesis will focus on two campaigns maintained by Google called *OSS-Fuzz* [22] and *FuzzBench* [16]: the first is a free platform that allows open-source developers to fuzz their programs autonomously by integrating with existing CI/CD pipelines and relying on the computing resources provided by the Google Cloud Service, while the second allows fuzzer developers to test and improve their tools against real-world benchmarks thanks to automated testing and reporting. The objective was to evaluate a selected group of the projects that have been implemented in these repositories using alternative approaches, trying to discover bugs that will be then securely reported and disclosed to their developers in the hope of having them fixed.

2.4.1 ClusterFuzz

Before introducing the frameworks mentioned above, it is also important to define the underlying infrastructure. The *ClusterFuzz Project* is a scalable fuzzing infrastructure with the objective of discovering security and stability issues in software through continuous coverage-guided fuzzing, it is also the main platform used by Google to test its products and the fuzzing back-end for *OSS-Fuzz*. As of May 2023, it discovered over 25.000 bugs in Google proprietary software (e.g. Chrome) and 36.000 bugs with OSS-Fuzz [10].

It is based on a highly scalable distributed system of VMs that performs fully automated fuzzing, bug triage, filing and closure of bug reports as well as providing performance statistics, and all operations are performed by two main components. The *App Engine* provides a web interface to the information collected during each fuzzing session, allowing the developers to easily access crashes, results, and other information. This is also where tests can be scheduled, which is done via `cron` jobs.

The *Fuzzing Bots Pool* is a cluster of VMs responsible for running the scheduled fuzzing sessions, and they perform the following operations:

- **fuzz:** runs a fuzzing session, configuration different for each fuzzer
- **progression:** checks if a testcase still reproduces or if it has been fixed
- **regression:** calculates the commits range in which a crash was introduced
- **minimize:** eliminates duplicate testcases from the input seeds
- **pruning:** minimize a corpus to the smallest size based on coverage information
- **analyze:** runs a manually uploaded testcase against a specific job to see if it crashes

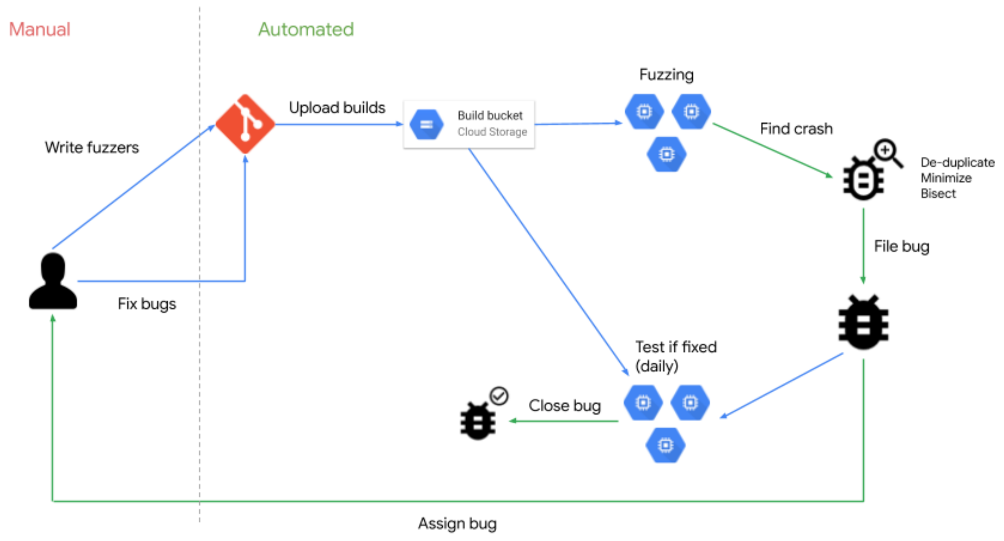


Figure 2.6. ClusterFuzz main architecture visualized [10]

Each VM performs these operations inside Docker instances created and uploaded by the developers, which are configured with all the tools and files necessary to correctly build and launch the fuzzing targets. Since some of these tasks are critical and should be treated as atomic operations, bots can be *preemptible* or *non-preemptible*: the former refers to a machine that can only run the "fuzz" task as it can be shut down at any moment, while the latter refers to a machine that is not expected to suddenly stop or crash and is therefore capable of performing all tasks.

2.4.2 OSS-Fuzz

The *OSS-Fuzz Project* [51] was founded in 2016 after the famous "Heartbleed" vulnerability was discovered in OpenSSL, one of the most popular open-source projects at the time for encrypting web traffic, as a response to provide developers with free fuzzing and private alerts services for their open-source projects. As of August 2023, it has helped identify and fix more than 10,000 vulnerabilities and 36,000 bugs in over 1000 projects [22].

While it was initially intended for languages that are not memory-safe (C/C++), it was later extended to provide support also for other popular languages such as Python, Go, Java, and Rust. Projects can be evaluated using several coverage-guided mutation-based fuzzing engines (such as LibFuzzer, AFL++ and Honggfuzz) in combination with Google Sanitizers (ASan, MSan and UBSan), while *ClusterFuzz* acts as the back-end and provides bug reporting functionalities.

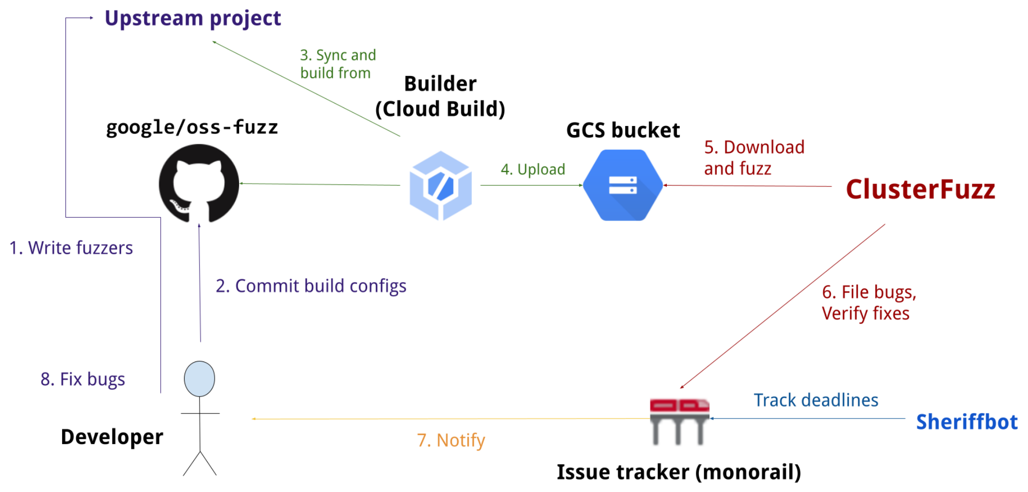


Figure 2.7. OSS-Fuzz main architecture visualized [22]

First, the maintainers of the open-source project create one or more "fuzz targets" that will be built and tested by ClusterFuzz system, where a "fuzz target" defines a function that takes an input (in this thesis, an array of bytes) and performs some operations on said input to test a specific API or program's functionality [36]. Developers are also expected to provide up-to-date corpus seeds, which will be used by the chosen fuzzers for cross-mutations and to improve the fuzz target's coverage. All these resources, along with instructions on how to build and fuzz the targets, are uploaded to a Google Cloud Service Bucket, a file-hosting service acting as a middle point between OSS-Fuzz and its fuzzing back-end ClusterFuzz.

Then, each fuzzing session lasts (at most) 6 hours, evenly distributed across all fuzz targets with a minimum of 10 minutes per fuzz target, and *ensemble fuzzing* is used when binaries are configured for multiple fuzzing engines (see Figure 2.5). The corpora used as inputs to the tests and the corpora produced as results of a successful fuzzing session (both also called *fuzzing queues*) are publicly available and stored in the project's Google Cloud for research purposes and *regression testing*, i.e. testing a program against old inputs to ensure that previously tested functionality still works as intended and that fixed bugs do not reappear.

Finally, any bug discovered is reported to the OSS-Fuzz issue tracker [21], which uses the metadata sent by ClusterFuzz to create its report. Developers have 3 ways of dealing with this situation: they can commit new changes to fix the bug (verified daily by ClusterFuzz before closing an issue), assign the tag "WontFix" to the bug and notify that it will not be solved, or simply ignore it altogether.

To be accepted by OSS-Fuzz, a project must be relevant and/or critical to the global IT infrastructure and it must have a significant user base. To check if your project is eligible, Google provides a mathematical formula to calculate its "Open-Source Project Criticality Score" [48] in a range between 0 *(least critical)* and 1 *(most critical)*:

$$CriticalityScore = \frac{1}{\sum \alpha_i} \sum_i \alpha_i \frac{\log(1 + S_i)}{\log(1 + \max(S_i, T_i))} \tag{2.1}$$

The evaluated parameters S_i are described below, while α_i refers to their "weight" and T_i to their maximum allowed value:

- **created-since** ($\alpha_i = 1$, $T_i = 120$): months since the project was created, older projects have a higher chance of being widely used or dependent on
- **updated-since** ($\alpha_i = -1$, $T_i = 120$): months since the project's last update, unmaintained projects with no recent commits have a higher chance of being less relied upon
- **contributor-count** ($\alpha_i = 2$, $T_i = 5000$): number of commits made by project contributors, involvement of different contributors indicates a project importance
- **org-count** ($\alpha_i = 1$, $T_i = 10$): number of distinct organizations contributing to the project, indicates cross-organization dependency
- **commit-frequency** ($\alpha_i = 1$, $T_i = 1000$): average number of commits per week over the last year, code that is constantly changing may be more susceptible to vulnerabilities
- **recent-releases-count** ($\alpha_i = 0.5$, $T_i = 26$): number of releases in the last year, frequent releases indicate user dependency
- **closed-issues-count** ($\alpha_i = 0.5$, $T_i = 5000$): number of issues closed in the last 90 days, indicates contributor involvement and focus on resolving user issues
- **updated-issues-count** ($\alpha_i = 0.5$, $T_i = 5000$): number of issues updated in the last 90 days, indicates high contributor involvement
- **comment-frequency** ($\alpha_i = 1$, $T_i = 15$): average number of comments per issue in the last 90 days, indicates user activity and dependence
- **dependents-count** ($\alpha_i = 2$, $T_i = 500000$): number of project mentions in the commit messages, indicates repository use

Only projects scoring a value ≥ 0.7 may be eligible to be integrated in the OSS-Fuzz campaign.

Assuming a project satisfies the score requirements, developers create the 3 following files and issue a "pull request" on the OSS-Fuzz repository, which are periodically checked and validated by bots before accepting/rejecting them.

Configuration. The "project.yaml' is a file that contains metadata regarding who owns the project, how to contact them when filing bug reports, and configuration parameters about the fuzzing sessions, needed by OSS-Fuzz to correctly provide its services. Follows a list some of the most common information reported:

- **homepage:** URL to the project homepage
- **language:** programming language used to write the project (C, C++, Go, Rust, Python, JVM languages, Swift)
- **primary_contacts:** list of email addresses that are automatically CC'd on crash reports and fuzzer statistics
- **main_repo:** path to the source repository where the project is hosted
- **vendor_ccs:** *optional*, list of email addresses of vendors who want access to bug reports
- **sanitizers:** *optional*, list of sanitizers available in the project (address, undefined, memory)
- **architectures:** *optional*, list of architectures that can build the project (supported only x86_64 and i386)
- **fuzzing_engines:** *optional*, list of available fuzzing engines (libfuzzer, afl and honggfuzz), if not specified "libfuzzer" is used
- **builds_per_day:** *optional*, number of times a project should be built per day, OSS-Fuzz allows up to 4 builds per day and builds once per day by default

Dockerfile. To avoid having developers create and maintain their own customized Docker containers and promoting a homogeneous and standard testing environment, OSS-Fuzz provides several "base images" based on Ubuntu 20.04 for each supported language containing the most common compilers and toolchains needed to build the project, along with some customized terminal commands useful to manipulate the image provided. Each project builds its own testing environment starting from the most appropriate base image, runs a series of "apt-get" and "git clone" commands to download and install all the necessary packages and dependencies, and finally pulls all the resources and fuzz targets from the project repository.

Builder. The "build.sh" is a script that runs inside the Docker container defined above and performs all the operations required to build the project and the fuzzing targets to be tested. To provide a more flexible and accessible build system, all base images are configured with several environment variables regarding directory locations, compilers, and compilation flags, allowing developers to easily re-target scripts with little effort.

To conclude, OSS-Fuzz follows a strict *bug disclosure policy* [24]. When a bug is discovered (assuming it is not already known or a duplicate), an automatic email is generated and sent to all email addresses specified in the "project.yaml" file, and an issue is opened in the issue tracker. This email contains the report generated by ClusterFuzz, as well as an estimate of the priority and severity of the bug discovered. From this moment, the issue will be publicly visible in 90 days or after the fix is released (whichever comes earlier), meaning that anyone will have access to the causing input as well as any other debugging information related to what happened and how to reproduce the bug. Before the deadline expires, developers may request a 14-day grace period if the patch is scheduled to be released on a specific day within this extended period, in which case the public disclosure will be delayed. In all cases, Google reserves the right to move deadlines forward or backward depending on the circumstances and severity of the findings.

2.4.3 FuzzBench

The *FuzzBench Project* [39] is a free service that provides fuzzer developers with several real-world benchmarks tested at Google-scale, comparing the results with other famous fuzzers (such as AFL and LibFuzzer), and allowing them to evaluate their performance thanks to daily reports for further improvement [16].

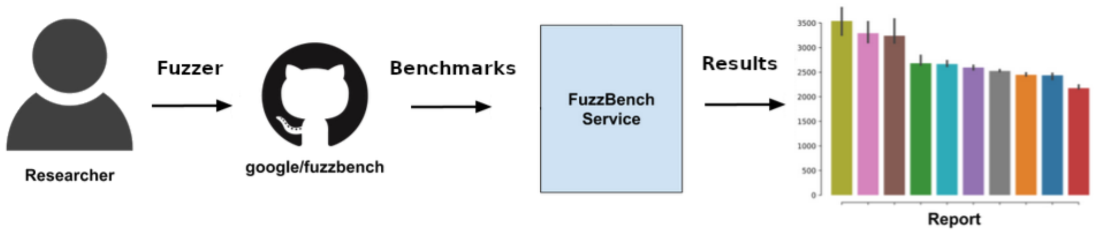


Figure 2.8. FuzzBench main architecture visualized [16]

Initially, the developer of a fuzzer integrates its tool within FuzzBench using a Dockerfile, containing all the resources necessary to build targets using its fuzzer and also defining the environment where all benchmarks will be executed. Similarly to OSS-Fuzz, this process is done via "pull requests", which are automatically revisioned and accepted by bots.

Then, two testing approaches are available: standard and OSS-Fuzz. In the former case, the benchmark is created by the developers themselves, requiring the definition of fuzz targets, build files, seed inputs and Docker images to correctly build, link the fuzzer to the targets and run the tests. In the latter case, the developers employ a fuzz target from a selection of OSS-Fuzz projects, maintained at a specific version known to contain bugs [15] and tested using a predefined set of seed inputs, ensuring that all tests are performed using the same test environment and initial inputs across many trials. This creates a reference point when tracking a fuzzer's performance over time, while also allowing the developers to test their product on a real-world scenario. When performing fuzzing, FuzzBench provides coverage-oriented or bug-oriented sessions, both spanning over 24 hours with each fuzzer running 20 trials on each individual benchmark selected.

At the end of each day, a performance report will be created highlighting the strengths and weaknesses of the fuzzer on each benchmark, comparing individual and overall results with other integrated fuzzers both in terms of bugs found and coverage achieved.

SBFT' 23

FuzzBench provides 21 open-source projects available as benchmarks, that are continuously tested by the integrated fuzzers. However, a preliminary analysis revealed a greater number of projects, including benchmarks not listed on the website [15]: after investigating the contents of this repository and its commit history, 7 additional projects were discovered, added for only a few months and then removed to never be restored.

These projects were related to a conference on the effectiveness of fuzzing tools called "Search-Based and Fuzzing Tools (SBFT) 2023" [34], where a tool competition was hosted using FuzzBench as the benchmarking tool: the idea was to allow developers to integrate their fuzzers into FuzzBench, run experiments locally on a provided sample benchmark to familiarize with real-world open-source projects and state-of-the-art fuzzers, and analyze the provided evaluation reports to visualize the effectiveness of their products. In particular, these 7 projects were used only during the final stages of the competition, ensuring that all participants performed the final evaluations on previously untested benchmarks.

The goal of this conference was to provide a way for developers to approach continuous fuzzing frameworks, contribute to the development of FuzzBench by integrating new and effective fuzzers, and provide real-world scenarios for developers to test their tools.

Chapter 3

Methodology

The aim of this thesis was to analyze common automated testing campaigns, specifically OSS-Fuzz and FuzzBench as they are among the most popular ones, to see if there were any overlooked bugs in the projects that were integrated: we define an overlooked bug as a bug for which a triggering input is produced by the fuzzing framework, but that is not successfully reported as such. The two frameworks considered provide different opportunities for bugs to be overlooked, as OSS-Fuzz uses an automated approach to how tests are scheduled, executed, and how bugs are reported, while FuzzBench requires manual scheduling and bug reporting: this, in turn, meant studying their workflow and examining how programs are tested, to verify how efficiently such frameworks are used by open-source developers. As previously mentioned, all relevant bugs were then appropriately reported.

In OSS-Fuzz, where fuzzing is performed automatically, we looked for cases where machines have failed: this could happen either due to a fault in the automation process or due to the integration choices made by the developers. In the former case, a fuzzer may succeed in producing a testcase that triggers a bug, but due to some problems in the implementation of the testing environment it is not successfully reported as such. In the latter, it is strictly related to the content of the *project.yaml* configuration file: different sanitizers look for different types of bugs, while each fuzzing engine uses its own set of strategies to test a program, producing results that may be completely different from the other provided fuzzers. Given this, OSS-Fuzz was analyzed by taking a small selection of projects and testing them locally on the latest version with a fixed combination of fuzzing engine and sanitizers, using the latest public fuzzing queue available on the project's repository as input corpus.

In FuzzBench, where projects are actively tested by several different fuzzers, we looked for bugs discovered in older versions that were not tested on newer ones, which obviously is a human error. As previously mentioned, this framework is based on a small selection of older (and bugged) versions of OSS-Fuzz projects that are continuously tested by several different fuzzers using a predefined testing environment, producing daily reports available both to the fuzzers' developers and the developers of the selected projects, along with public access to the standard corpora used to perform the tests, the results and the crashes found (if any). It is then responsibility of the project developers to analyze and fix these bugs, although this does not always happen. Given this, FuzzBench was analyzed by testing all available benchmarks and the ones from the SBFT '23 conference (see 2.4.3), building the projects to their latest OSS-Fuzz version using AFL as fuzzing engine, ASan+UBSan as sanitizers, and using as input corpus all the available crashes for each benchmark found in all fuzzing session that were publicly accessible between 2020 and October 2024 (at the moment of testing).

3.1 Setting up the Environment

All tests were performed on two separate machines with similar specs, both equipped with personal installations of Ubuntu 22.04 LTS already run-in and used, employing the following tools:

- *Docker* [2]: virtualization tool used to run the different containers needed by OSS-Fuzz to create the environments where each project was built and tested
- *Valgrind* [45][44]: dynamic binary instrumentation (DBI) framework with tools that perform analysis, profiling, and management of a program during its execution, used specifically for the "Memcheck" tool when looking for memory-related bugs in fuzz targets built without sanitizers
- *Python* (v. 3.10.12) [7]: needed by OSS-Fuzz to provide its functionalities and used to create several scripts to perform information and web scraping, reports analysis and bug deduplication
- *gsutil* [20]: command-line tool suite provided by Google to remotely access data stored on Google Cloud Service from your local machine, used to analyze FuzzBench experiments and download other resources
- *Google Chrome/Development Driver* [19]: used during the information scraping phase of this work, discussed in section 3.2.1

Regarding *OSS-Fuzz*, most of the preparation was done using its GitHub repository, which was cloned locally on both machines. To provide its services, OSS-Fuzz uses several Python scripts that can be invoked from the command line with appropriate arguments: these commands can be used to update the repository and the files used to build each project, update the base images and each project image, build the project image and its fuzzers, and finally download other resources such as reports and publicly available corpora.

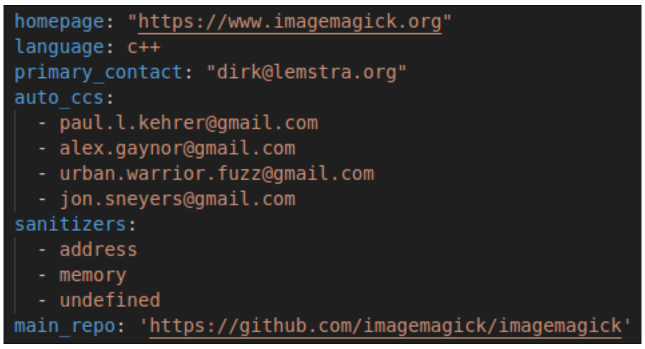
Regarding *FuzzBench*, most of the preparation was done by incorporating the *gsutil* suite in a Python script that performed web scraping and downloaded content from its Google Cloud bucket, specifically from the location where all experiments results are collected.

All tests were run by providing Docker with unlimited access to the CPU cores and disk space, while RAM was limited to 16GB, to ensure that no input may cause an out-of-memory situation that could result in a complete crash of the testing machines.

3.2 OSS-Fuzz

3.2.1 Selecting the Projects

At the time of writing, the OSS-Fuzz campaign includes over 1000 projects that are actively being fuzzed and tested, but rebuilding and testing all of them locally would require not only extensive knowledge of many programming languages, but also time and skills to fix potential build errors left by developers' negligence. For this reason, this work focused exclusively on projects written in C/C++, as these languages are widely known to be prone to human error. Then, to further narrow down the analysis, I identified 5 different categories based on the number of sanitizers selected by the developers for testing, keeping ASan as the reference due to its popularity and efficacy. Finally, I ordered each set by "highest number of bugs issued" using the OSS-Fuzz bug tracker and tested these lists from top to bottom until I had 5 projects for each category that were building and fuzzing correctly.



```
homepage: "https://www.imagemagick.org"
language: c++
primary_contact: "dirk@lemstra.org"
auto_ccs:
  - paul.l.kehrer@gmail.com
  - alex.gaynor@gmail.com
  - urban.warrior.fuzz@gmail.com
  - jon.sneyers@gmail.com
sanitizers:
  - address
  - memory
  - undefined
main_repo: 'https://github.com/imagemagick/imagemagick'
```

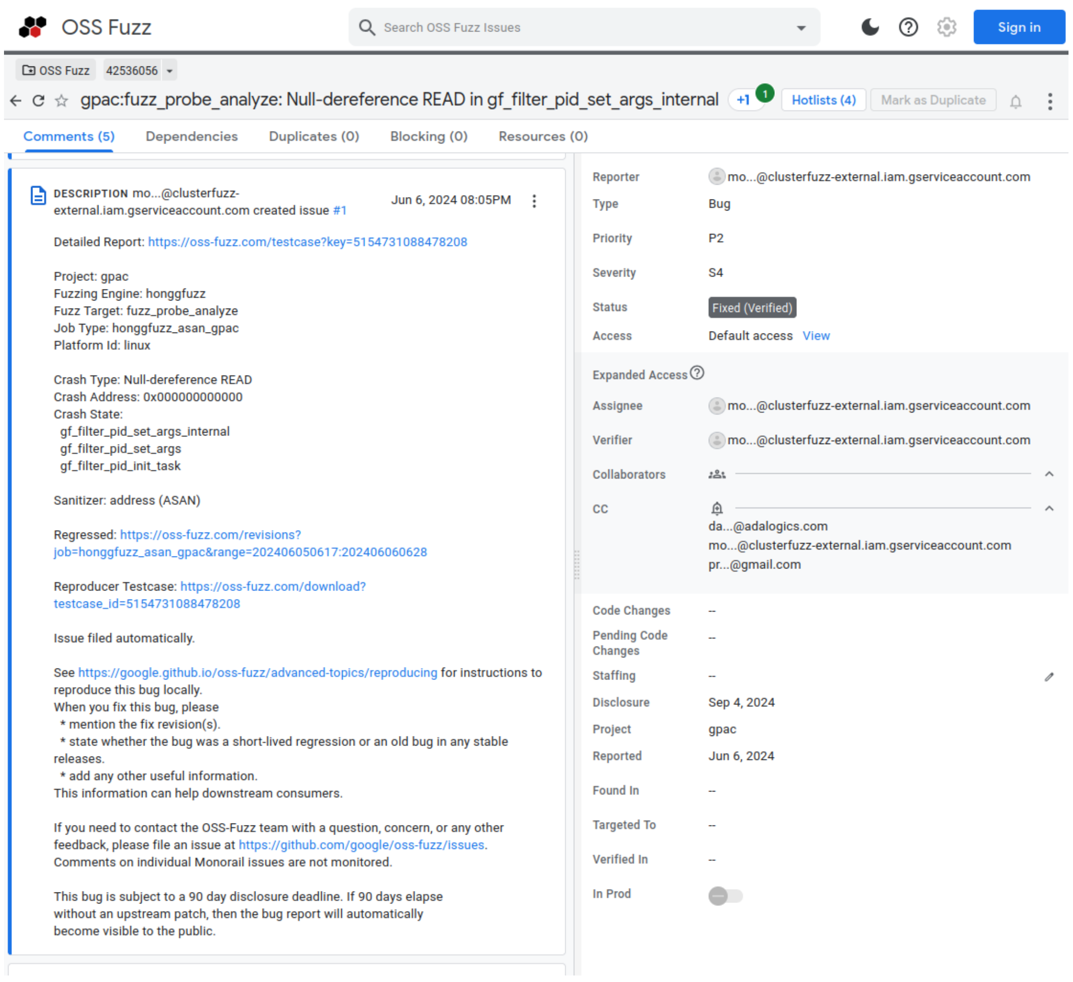
Figure 3.1. Example of content from a project.yaml

The first step in this process was to extract the list of all projects written in C/C++ and divide them according to the sanitizers used, and this was done by performing a preliminary analysis of the *project.yaml* configuration file present inside each project's directory. To retrieve this information, I wrote a simple Python script that iteratively explored each project's directory looking for the aforementioned configuration file, opened the file (assuming it was found) and scanned each line looking for the "language: c" string and the keywords "address", "memory" and "undefined", eventually saving the name of each project in the appropriate list.

This yielded a total of 524 projects out of 1277 written using C/C++, with:

- 238 projects using all sanitizers when fuzzing
- 22 projects using ASan+MSan when fuzzing
- 62 projects using ASan+UBSan when fuzzing
- 46 projects using only ASan when fuzzing
- 156 projects not using any sanitizer when fuzzing

The next step was to determine the number of bugs already found for each project, and this required a thorough analysis of the “OSS-Fuzz Issue Tracker” website [21]. At the time of writing, the issue tracker platform has changed from “Monorail” to “Google Sites”, so most of the work described here may no longer work as intended.



DESCRIPTION mo...@clusterfuzz-external.iam.gserviceaccount.com created issue #1 Jun 6, 2024 08:05PM

Detailed Report: https://oss-fuzz.com/testcase?key=5154731088478208

Project: gpac
Fuzzing Engine: honggfuzz
Fuzz Target: fuzz_probe_analyze
Job Type: honggfuzz_asan_gpac
Platform Id: linux

Crash Type: Null-dereference READ
Crash Address: 0x000000000000
Crash State:
 gf_filter_pid_set_args_internal
 gf_filter_pid_set_args
 gf_filter_pid_init_task

Sanitizer: address (ASAN)

Regressed: https://oss-fuzz.com/revisions?job=honggfuzz_asan_gpac&range=202406050617:202406060628

Reproducer Testcase: https://oss-fuzz.com/download?testcase_id=5154731088478208

Issue filed automatically.

See https://google.github.io/oss-fuzz/advanced-topics/reproducing for instructions to reproduce this bug locally.
When you fix this bug, please
 * mention the fix revision(s).
 * state whether the bug was a short-lived regression or an old bug in any stable releases.
 * add any other useful information.
This information can help downstream consumers.

If you need to contact the OSS-Fuzz team with a question, concern, or any other feedback, please file an issue at https://github.com/google/oss-fuzz/issues.
Comments on individual Monorail issues are not monitored.

This bug is subject to a 90 day disclosure deadline. If 90 days elapse without an upstream patch, then the bug report will automatically become visible to the public.

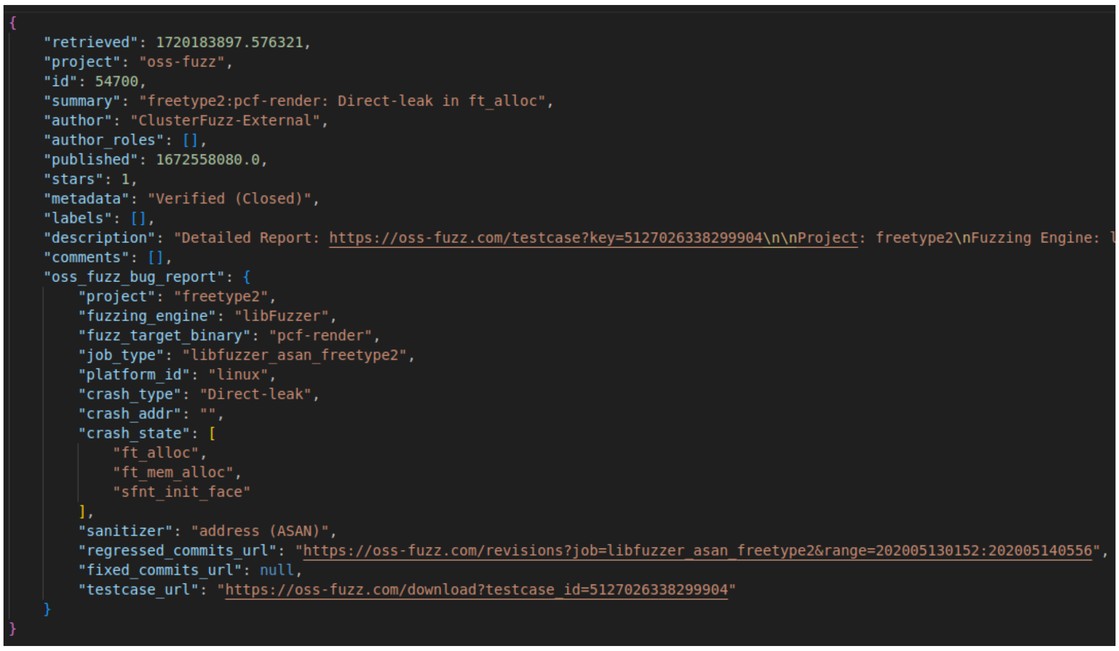
Reporter	mo...@clusterfuzz-external.iam.gserviceaccount.com
Type	Bug
Priority	P2
Severity	S4
Status	Fixed (Verified)
Access	Default access View
Expanded Access	
Assignee	mo...@clusterfuzz-external.iam.gserviceaccount.com
Verifier	mo...@clusterfuzz-external.iam.gserviceaccount.com
Collaborators	
CC	da...@adalogics.com mo...@clusterfuzz-external.iam.gserviceaccount.com pr...@gmail.com
Code Changes	--
Pending Code Changes	--
Staffing	--
Disclosure	Sep 4, 2024
Project	gpac
Reported	Jun 6, 2024
Found In	--
Targeted To	--
Verified In	--
In Prod	

Figure 3.2. Example of bug report [21]

Since Monorail APIs could only be accessed by the developers of projects already integrated into OSS-Fuzz and there were no files that could be used for offline analysis, I had to perform web scraping on the individual issues from their web reports. To do this, I used as reference a GitHub repository written by Zhen Yu Ding called “Monorail Scraper” [64], a Python script for scraping and retrieving data from Monorail-based platforms, which also includes functionalities for ClusterFuzz-generated OSS-Fuzz issues. The tool relies on Google Chrome and its development tool called “ChromeDriver” [19], an autonomous web server implementing the “W3C WebDriver” [59], which in turn provides a remote interface to control user-agents and a set of interfaces to perform analysis and manipulation of DOM elements. Essentially, all these components combined allow the user to write scripts that, in turn, instruct the Google Chrome browser to visit a particular web page, analyze its DOM elements, and possibly perform some interaction with it.

The produced Python web-scraping script requested a range of "report IDs" to be retrieved, then opened a new Google Chrome instance and performed a connection to a specifically constructed link on the Monorail website, attempting to reconnect only once if the first fails. If the resulting DOM displayed a login form, it meant that the requested bug wass still in the disclosure window, in which case the next ID was analyzed. Assuming that the requested report was publicly available, the DOM was scanned for key information, finally stored in JSON files.

This analysis was performed on all bugs between 2023-01-01 and 2024-06-31, for a total of 11743 collected reports.



```json
{
    "retrieved": 1720183897.576321,
    "project": "oss-fuzz",
    "id": 54700,
    "summary": "freetype2:pcf-render: Direct-leak in ft_alloc",
    "author": "ClusterFuzz-External",
    "author_roles": [],
    "published": 1672558080.0,
    "stars": 1,
    "metadata": "Verified (Closed)",
    "labels": [],
    "description": "Detailed Report: https://oss-fuzz.com/testcase?key=5127026338299904\n\nProject: freetype2\nFuzzing Engine: l
    "comments": [],
    "oss_fuzz_bug_report": {
        "project": "freetype2",
        "fuzzing_engine": "libFuzzer",
        "fuzz_target_binary": "pcf-render",
        "job_type": "libfuzzer_asan_freetype2",
        "platform_id": "linux",
        "crash_type": "Direct-leak",
        "crash_addr": "",
        "crash_state": [
            "ft_alloc",
            "ft_mem_alloc",
            "sfnt_init_face"
        ],
        "sanitizer": "address (ASAN)",
        "regressed_commits_url": "https://oss-fuzz.com/revisions?job=libfuzzer_asan_freetype2&range=202005130152:202005140556",
        "fixed_commits_url": null,
        "testcase_url": "https://oss-fuzz.com/download?testcase_id=5127026338299904"
    }
}
```

Figure 3.3. Example of information collected from a report

The second to last step was to analyze the JSON files and make a list of the most buggy projects for each category, which was done by writing a simple Python script that takes as input these files and analyzes the information fields collected, shown in Figure 3.3. First, it checked the *"metadata"* field for values such as "WontFix", "Duplicate" or "Invalid": the first means that the developers themselves tagged that specific bug as non-relevant and therefore will not be addressed in the future, the second refers to a report for a bug that has been already issued but was triggered by a different testcase, while the last one means that the reported bug could not be reliably reproduced using the provided testcase. Then, looked at the *"description"* field for manual reports, which have been ignored as the information they contained was not always written according to the same standard used by ClusterFuzz and therefore not useful towards the collection of the information required. Assuming the report analyzed was valid and generated by ClusterFuzz, it retrieved the project name from the *"oss_fuzz_bug_report"* fields, using a dictionary key-value to keep track of the number of bugs reported for each project.

The final step was to analyze all the bugs reported by each project individually and determine which fuzz target produced the highest number of reports. Given the previous script, I extended it to take the project name as input, so that the parsing of the JSON file focused only on reports for that particular project, and the dictionary key-value was now used to keep track of the number of bugs produced by each fuzzing target binary.

All the aforementioned work produced Table 3.1, which shows all the selected projects and their respective most bugged binaries tested:

Project	**Sanitizer**	**Tested binary**
binutils	ALL	fuzz_objdump_safe
harfbuzz	ALL	hb-subset-fuzzer
imagemagick	ALL	encoder_heic_fuzzer
libxml2	ALL	valid
skia	ALL	skruntimeeffect
gpsd	A+M	FuzzPacket
libyang	A+M	lyd_parse_mem_json
llvm	A+M	clang-fuzzer
openjpeg	A+M	opj_decompress_fuzz_J2K
wasmedge	A+M	wasmedge-fuzztool
cairo	A+UB	svg-render-fuzzer
clamav	A+UB	clamav_dbload_YARA_fuzzer
freerdp	A+UB	TestFuzzCoreClient
tarantool	A+UB	luaL_loadbuffer_fuzzer
vlc	A+UB	vlc-demux-dec-libfuzzer
fwupd	ASan only	uswid_fuzzer
glslang	ASan only	compile_fuzzer
inchi	ASan only	inchi_input_fuzzer
radare2	ASan only	ia_fuzz
zeek	ASan only	zeek-ftp-fuzzer
fluent-bit	NONE	flb-it-fuzz-cmetric_decode_fuzz_OSSFUZZ
gpac	NONE	fuzz_probe_analyze
libdwarf	NONE	fuzz_debug_str
libredwg	NONE	llvmfuzz
serenity	NONE	FuzzJs

Table 3.1. Open-source projects tested and fuzzing binaries analyzed

3.2.2 Testing with OSS-Fuzz

The OSS-Fuzz repository contains several Python scripts to build and test the available projects, as well as debugging and reproducing crashes and bugs. Most of the tools used in this work are provided by the "helper.py" script, which I used to download a project's Docker image, build the fuzzers and download the latest public corpora made available by the developers, using the following commands:

```
$ python3 helper.py pull_images

$ python3 helper.py build_image {project_name}

$ python3 helper.py build_fuzzers {project_name}
    --sanitizer={address(default),memory,undefined,none}
        --engine={libfuzzer(default),afl, honggfuzz, centipede}

$ python3 helper.py download_corpora
    --project={project_name} --fuzz-target={binary_name}
```

The `pull_images` argument connects to OSS-Fuzz's Google Bucket to download and update all the Docker "base images" on your local machine, which are needed by all projects to create their testing environments. The `build_image` argument takes as input the name of a project and builds its Dockerfile, creating a new image on your local machine that will be later used to perform fuzzing. During this process, all dependencies and resources needed to correctly compile the fuzzers are downloaded and installed, including the main *build.sh* script. It was used in conjunction with the previous command and daily `git pull` on the OSS-Fuzz repository, to make sure that I was always building with the latest versions. The `build_fuzzers` argument takes as input the name of a project, a list of possible sanitizers, and a list of possible fuzzing engines to be used during the compilation of the fuzz targets. Although this command accepts only one sanitizer and fuzzing engine at a time, it is possible to mix them by acting on some environment variables provided by the base images. This command also acts as a "wrapper" for a much more complex Docker command, that loads the project's Docker image using some specific environment variables and invokes the execution of the *build.sh* script. The `download_corpora` argument takes as input a project name and a fuzz target, it then connects to the project's Google Bucket and downloads the latest public corpus for the provided fuzz target.

After executing these commands, three new directories are created: `out` contains the project's directory where all built files were saved, including libraries, fuzz targets and other files created by the selected fuzzer, `work` acts as a temporary location to store intermediate files during the building process and the fuzzing sessions, and `corpus` contains the downloaded corpora stored as a zip file.

To prepare the tests, I was tasked with building the chosen fuzz targets using AFL++ (current state-of-the-art fuzzer) and with all possible sanitizers, meaning that each project was compiled 4 times: with ASan only, with MSan only, with UBSan only, and without any sanitizer.

The tests were performed inside each project's Docker image, created and configured using the following command:

```
$ docker run --rm --privileged
    --platform linux/amd64 --memory=16g
    -v /oss-fuzz/build/out/{project_name}/:/out/
    -v /oss-fuzz/build/corpus/{project_name}/:/corpus/
    -v /home/zio-saba/Scrivania/TESI/logfiles/:/logfiles/
    -it  gcr.io/oss-fuzz/{project_name} /bin/bash
```

The first parameters are needed to create a privileged instance of Docker, specify the running platform on which the fuzz targets will be tested and limit memory usage inside the container. The arguments starting with `-v` are used to create a shared directory, i.e. linking a local directory to a virtual one created inside the container: this was necessary to make sure that I could access the resources stored locally on my machine (i.e. fuzz targets, libraries and their corpus) from inside the Docker container. The last line invokes the project image to load as well as making it interactive by spawning a `/bin/bash` process.

Once the Docker image has been loaded and ready to use, some final adjustments were made to perform the tests, here we mention a few of them:

- all tests performed on fuzz targets built with MSan required the sanitizer's libraries to be copied in some specific locations, using the following commands:

  ```
  $ cp -R /usr/msan/lib/* /usr/local/lib/x86_64-unknown-linux-gnu/
  $ cp -R /usr/msan/include/* /usr/local/include
  ```

- all tests performed on fuzz targets built without sanitizers relied on Valgrind to perform binary analysis and profiling, installed using the following command:

  ```
  $ sudo apt install gdb valgrind
  ```

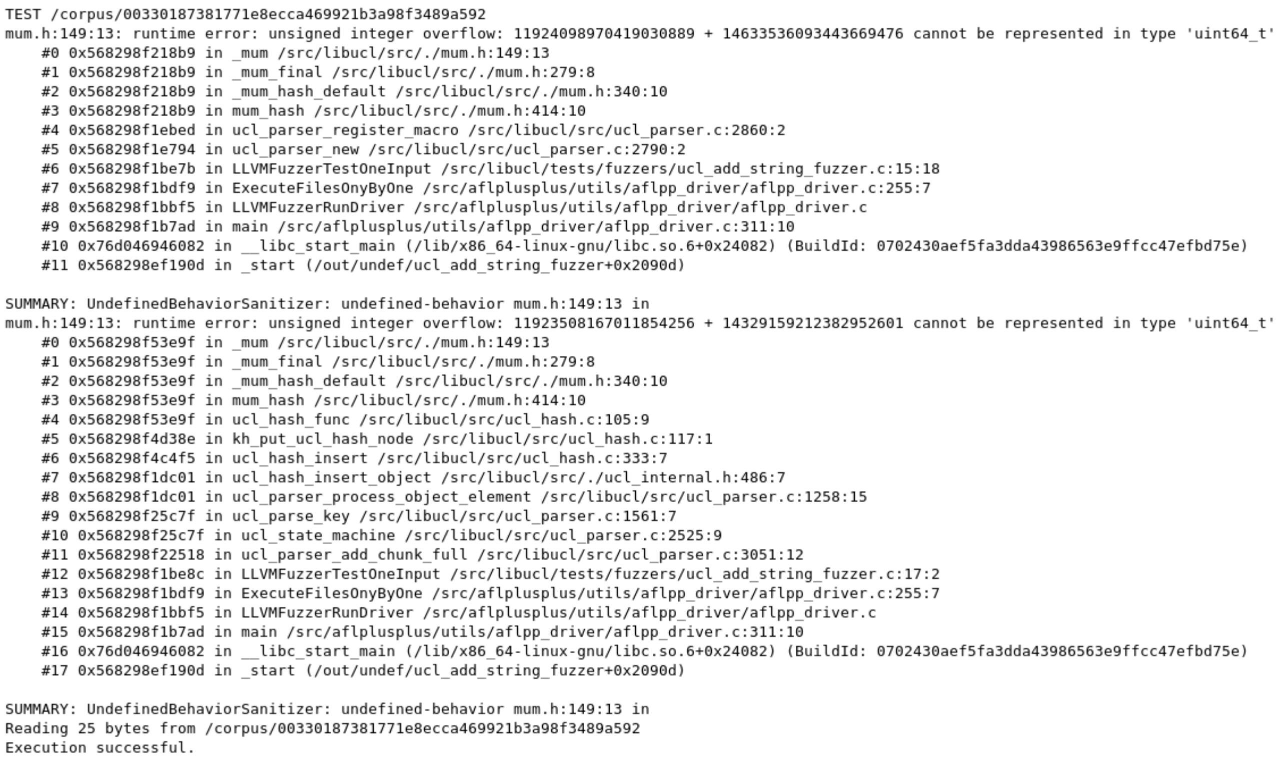
- when modifying the source files for compile errors, the `compile` command was used to start the build process for the fuzz targets without leaving the Docker image

- sometimes, the `apt` tool was not available in a specific Docker container, because all base images provided by OSS-Fuzz contain a minimal installation with only some key packages that are usually enough to compile programs, such as compiler, assembler, text editors and standard libraries: in such cases, the `unminimize` command was executed, which essentially “unpacks” the container and reverts it to a standard Ubuntu image, reinstalling all the default packages as well as standard additional tools

When testing fuzz targets built with ASan, MSan or UBSan, I used the following command:

```
$ for i in /corpus/*; do
    echo "TEST" $i;
    echo "TEST" $i >> /logfiles/PROJECT_SANITIZER-NAME.log;
    ./{fuzz_target} $i &>> /logfiles/PROJECT_SANITIZER-NAME.log;
  done
```

The shell `for` construct scans all files found in the “corpus” directory, then prints the name of the current testcase on the terminal for monitoring purposes, and finally logs the name of the current testcase and the results of the fuzz target executed on that particular testcase on a log file.



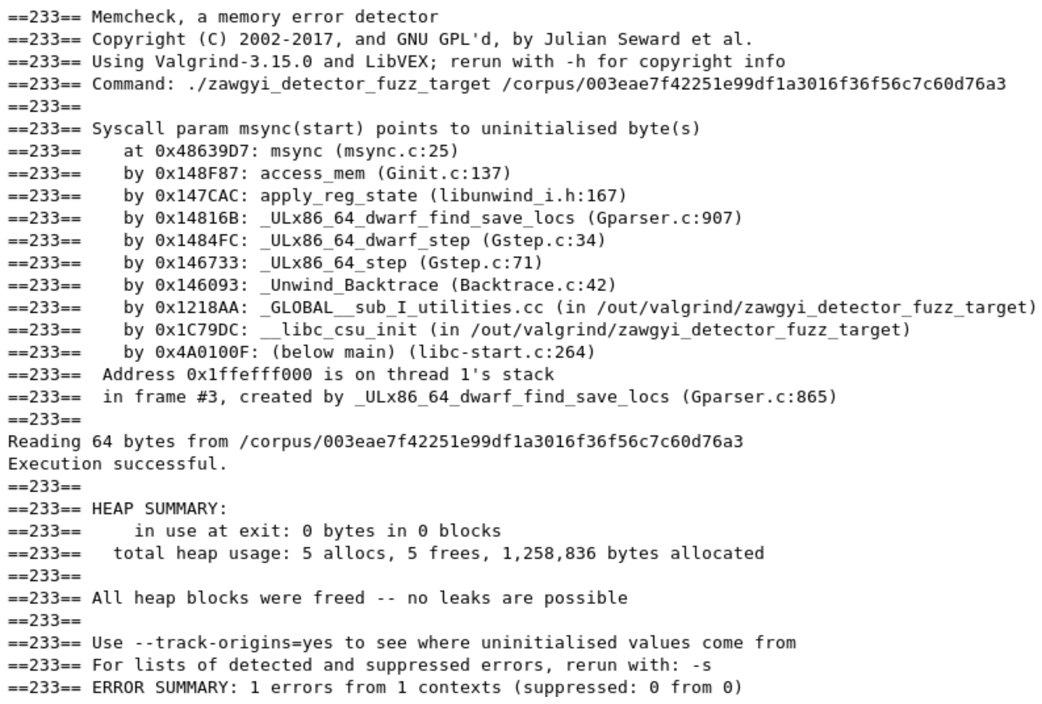
```
TEST /corpus/00330187381771e8ecca469921b3a98f3489a592
mum.h:149:13: runtime error: unsigned integer overflow: 11924098970419030889 + 14633536093443669476 cannot be represented in type 'uint64_t'
    #0 0x568298f218b9 in _mum /src/libucl/src/./mum.h:149:13
    #1 0x568298f218b9 in _mum_final /src/libucl/src/./mum.h:279:8
    #2 0x568298f218b9 in _mum_hash_default /src/libucl/src/./mum.h:340:10
    #3 0x568298f218b9 in mum_hash /src/libucl/src/./mum.h:414:10
    #4 0x568298f1ebed in ucl_parser_register_macro /src/libucl/src/ucl_parser.c:2860:2
    #5 0x568298f1e794 in ucl_parser_new /src/libucl/src/ucl_parser.c:2790:2
    #6 0x568298f1be7b in LLVMFuzzerTestOneInput /src/libucl/tests/fuzzers/ucl_add_string_fuzzer.c:15:18
    #7 0x568298f1bdf9 in ExecuteFilesOnyByOne /src/aflplusplus/utils/aflpp_driver/aflpp_driver.c:255:7
    #8 0x568298f1bbf5 in LLVMFuzzerRunDriver /src/aflplusplus/utils/aflpp_driver/aflpp_driver.c
    #9 0x568298f1b7ad in main /src/aflplusplus/utils/aflpp_driver/aflpp_driver.c:311:10
    #10 0x76d046946082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: 0702430aef5fa3dda43986563e9ffcc47efbd75e)
    #11 0x568298ef190d in _start (/out/undef/ucl_add_string_fuzzer+0x2090d)

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior mum.h:149:13 in
mum.h:149:13: runtime error: unsigned integer overflow: 11923508167011854256 + 14329159212382952601 cannot be represented in type 'uint64_t'
    #0 0x568298f53e9f in _mum /src/libucl/src/./mum.h:149:13
    #1 0x568298f53e9f in _mum_final /src/libucl/src/./mum.h:279:8
    #2 0x568298f53e9f in _mum_hash_default /src/libucl/src/./mum.h:340:10
    #3 0x568298f53e9f in mum_hash /src/libucl/src/./mum.h:414:10
    #4 0x568298f53e9f in ucl_hash_func /src/libucl/src/ucl_hash.c:105:9
    #5 0x568298f4d38e in kh_put_ucl_hash_node /src/libucl/src/ucl_hash.c:117:1
    #6 0x568298f4c4f5 in ucl_hash_insert /src/libucl/src/ucl_hash.c:333:7
    #7 0x568298f1dc01 in ucl_hash_insert_object /src/libucl/src/./ucl_internal.h:486:7
    #8 0x568298f1dc01 in ucl_parser_process_object_element /src/libucl/src/ucl_parser.c:1258:15
    #9 0x568298f25c7f in ucl_parse_key /src/libucl/src/ucl_parser.c:1561:7
    #10 0x568298f25c7f in ucl_state_machine /src/libucl/src/ucl_parser.c:2525:9
    #11 0x568298f22518 in ucl_parser_add_chunk_full /src/libucl/src/ucl_parser.c:3051:12
    #12 0x568298f1be8c in LLVMFuzzerTestOneInput /src/libucl/tests/fuzzers/ucl_add_string_fuzzer.c:17:2
    #13 0x568298f1bdf9 in ExecuteFilesOnyByOne /src/aflplusplus/utils/aflpp_driver/aflpp_driver.c:255:7
    #14 0x568298f1bbf5 in LLVMFuzzerRunDriver /src/aflplusplus/utils/aflpp_driver/aflpp_driver.c
    #15 0x568298f1b7ad in main /src/aflplusplus/utils/aflpp_driver/aflpp_driver.c:311:10
    #16 0x76d046946082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082) (BuildId: 0702430aef5fa3dda43986563e9ffcc47efbd75e)
    #17 0x568298ef190d in _start (/out/undef/ucl_add_string_fuzzer+0x2090d)

SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior mum.h:149:13 in
Reading 25 bytes from /corpus/00330187381771e8ecca469921b3a98f3489a592
Execution successful.
```

Figure 3.4. Example of integer-overflow bug reported by UBSan

When testing fuzz targets built without any sanitizers, I used Valgrind to analyze and profile the binary's execution, using the following command:

```
$ for i in /corpus/*; do
    echo "TEST" $i;
    valgrind --log-fd=9 9>>/logfiles/PROJECT_valgrind.log
        ./{fuzz_target} $i >> /logfiles/PROJECT_valgrind.log
        && echo -e "\n\n" >> /logfiles/PROJECT_valgrind.log;
  done
```

Similarly to before, the shell `for` construct scans all files found in the "corpus" directory and prints the name of the current testcase on the terminal for monitoring purposes, then Valgrind is invoked with a custom file descriptor to redirect `STDERR` in the log file, and the name of the current testcase and the results of the fuzz target executed on that particular testcase are logged on a log file.



```
==233== Memcheck, a memory error detector
==233== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==233== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==233== Command: ./zawgyi_detector_fuzz_target /corpus/003eae7f42251e99df1a3016f36f56c7c60d76a3
==233==
==233== Syscall param msync(start) points to uninitialised byte(s)
==233==    at 0x48639D7: msync (msync.c:25)
==233==    by 0x148F87: access_mem (Ginit.c:137)
==233==    by 0x147CAC: apply_reg_state (libunwind_i.h:167)
==233==    by 0x14816B: _ULx86_64_dwarf_find_save_locs (Gparser.c:907)
==233==    by 0x1484FC: _ULx86_64_dwarf_step (Gstep.c:34)
==233==    by 0x146733: _ULx86_64_step (Gstep.c:71)
==233==    by 0x146093: _Unwind_Backtrace (Backtrace.c:42)
==233==    by 0x1218AA: _GLOBAL__sub_I_utilities.cc (in /out/valgrind/zawgyi_detector_fuzz_target)
==233==    by 0x1C79DC: __libc_csu_init (in /out/valgrind/zawgyi_detector_fuzz_target)
==233==    by 0x4A0100F: (below main) (libc-start.c:264)
==233==  Address 0x1ffefff000 is on thread 1's stack
==233==  in frame #3, created by _ULx86_64_dwarf_find_save_locs (Gparser.c:865)
==233==
Reading 64 bytes from /corpus/003eae7f42251e99df1a3016f36f56c7c60d76a3
Execution successful.
==233==
==233== HEAP SUMMARY:
==233==     in use at exit: 0 bytes in 0 blocks
==233==   total heap usage: 5 allocs, 5 frees, 1,258,836 bytes allocated
==233==
==233== All heap blocks were freed -- no leaks are possible
==233==
==233== Use --track-origins=yes to see where uninitialised values come from
==233== For lists of detected and suppressed errors, rerun with: -s
==233== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Figure 3.5. Example of use-of-uninitialized-memory (UUM) bug reported by Valgrind

However, not all projects successfully built on their first attempt. In most cases, the building process failed due to missing libraries: for example, libraries like `pthread` and `math` were often automatically added by the sanitizers. Other times, libraries were not being correctly linked and/or were missing crucial compiling flags, like `-ldl` and `-lz`. To fix these problems, I had to manually modify the source files and compile the fuzz targets from inside the Docker container, using the following command:

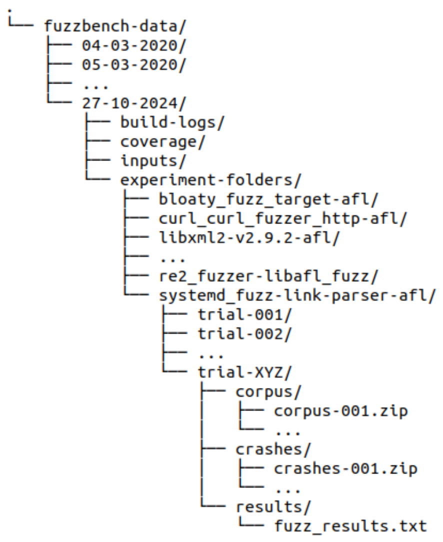
```
$ docker run --rm --privileged
    --platform linux/amd64 --memory=16g
    -e PROJECT_NAME={project_name} -e HELPER=True
    -e FUZZING_LANGUAGE=c++
    -e FUZZING_ENGINE=afl
    -e SANITIZER={address,memory,undefined,none}
    -v /oss-fuzz/build/out/{project_name}/:/out/
    -v /oss-fuzz/build/work/{project_name}/:/work/
    -it  gcr.io/oss-fuzz/{project_name} /bin/bash
```

Similarly to before, the first parameters are needed to create a privileged instance of Docker, specify the running platform on which the fuzz targets will be tested and limit memory usage, the arguments starting with `-v` are used to create a shared directory in the Docker container and the last line invokes the project image to load as well as making it interactive by spawning a `/bin/bash` process. The novelty lies in the arguments starting with `-e`, which can be used to override the environment variables provided by the Docker image: these variables can be used by the source files to compile a program using the most appropriate compile flags, fuzzing engine and sanitizers, and they can be easily modified by the user when creating the Docker image to easily re-target the compilation with little to none effort.

3.3 FuzzBench

3.3.1 Selecting the Projects

The FuzzBench campaign started in 2020 and has been performing tests almost daily for the past 4 years, providing valuable information to both fuzzers developers and the developers of the selected projects that have been integrated as benchmarks.



```
.
└── fuzzbench-data/
    ├── 04-03-2020/
    ├── 05-03-2020/
    ├── ...
    └── 27-10-2024/
        ├── build-logs/
        ├── coverage/
        ├── inputs/
        └── experiment-folders/
            ├── bloaty_fuzz_target-afl/
            ├── curl_curl_fuzzer_http-afl/
            ├── libxml2-v2.9.2-afl/
            ├── ...
            ├── re2_fuzzer-libafl_fuzz/
            └── systemd_fuzz-link-parser-afl/
                ├── trial-001/
                ├── trial-002/
                ├── ...
                └── trial-XYZ/
                    ├── corpus/
                    │   ├── corpus-001.zip
                    │   └── ...
                    ├── crashes/
                    │   ├── crashes-001.zip
                    │   └── ...
                    └── results/
                        └── fuzz_results.txt
```

Figure 3.6. Simplified structure of the FuzzBench's Google Cloud directory tree

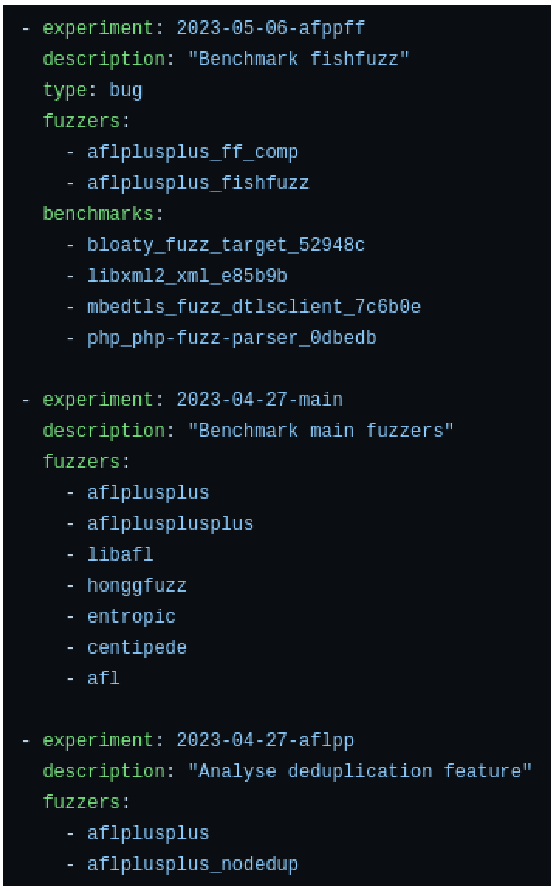
All data on FuzzBench is grouped by test date, and each test set is composed by several key information:

- `build-logs`: contains the logs generated when building the fuzz targets
- `coverage`: contains information related to the coverage achieved by each fuzzer on the different projects tested that day
- `input`: contains the binaries executed to compile the different fuzz targets
- `experiment-folders`: contains all the data related to the testing sessions

Following the experiments, each project has its own folder that specifies the name of the project, the fuzz target tested, the fuzzing engine used and sometimes also the objective of the session (bugs, coverage, correctness). Each project then undergoes several fuzzing sessions, identified by different `trial` folders, providing all the information that was relevant to this work:

- `corpus`: contains several corpora stored as zip files.
- `crashes`: contains all the crashes found in each trial as a separate zip file
- `results`: contains the cumulative log of all fuzzing sessions

The objective of this analysis was to build the latest version of several experiments, download all the available `crashes` for each one of them and verify whether they have been fixed or not. However, a preliminary analysis yielded over 100 million possible zip files to download and test, which would require several months of work by itself. As previously mentioned, FuzzBench provides coverage-oriented or bug-oriented fuzzing, therefore I focused on bug-oriented experiments for the analysis. To do this, I found the *experiment-requests.yaml* configuration file [13] inside the FuzzBench repository, which contained all information related to the type of tests conducted, the projects tested as well as all the fuzzing engines used:



```yaml
- experiment: 2023-05-06-afppff
  description: "Benchmark fishfuzz"
  type: bug
  fuzzers:
    - aflplusplus_ff_comp
    - aflplusplus_fishfuzz
  benchmarks:
    - bloaty_fuzz_target_52948c
    - libxml2_xml_e85b9b
    - mbedtls_fuzz_dtlsclient_7c6b0e
    - php_php-fuzz-parser_0dbedb

- experiment: 2023-04-27-main
  description: "Benchmark main fuzzers"
  fuzzers:
    - aflplusplus
    - aflplusplusplus
    - libafl
    - honggfuzz
    - entropic
    - centipede
    - afl

- experiment: 2023-04-27-aflpp
  description: "Analyse deduplication feature"
  fuzzers:
    - aflplusplus
    - aflplusplus_nodedup
```

Figure 3.7. Excerpt from the experiment configuration file

Similarly to the process shown in section 3.2.1, I wrote a simple Python script taking as input the above file and storing the name of all those experiments containing the string “type: bug”. Unfortunately, this file only contained information on all experiments between 2020 and 2023, forcing me to later retrieve all the experiments performed in 2024.

Regarding the selection of the projects and fuzz targets to test, while the initial idea was to perform the analysis only on the benchmarks provided by FuzzBench, it was later decided to also include the special benchmarks added during the “SBFT ’23” conference (refer to 2.4.3) due to the overwhelmingly amount of bugs discovered when performing preliminary tests on them.

Finally, to retrieve all possible crashes from the bug-oriented experiments performed in 2020-2023 and all experiments from 2024, I used the *gsutil* suite provided by Google, which works similarly to the `ls`, `mv` and `cp` command of the Linux shell, except that it takes as argument the URL of a Google Cloud resource. To construct the URL of all the resources needed, I wrote a Python script that essentially performs horizontal scraping over the directory tree shown before (see Figure 3.6), storing at each intermediate step the link of the next resource to list using the `gsutil ls` command. This analysis yielded 971656 zip files, for a total of 9 million crashes to test.

All the aforementioned work produced Table 3.2, which shows the standard FuzzBench benchmarks (above) and the SBFT'23 benchmarks (below) along with their respective binaries tested:

Project	**Tested binary**
bloaty	fuzz_target
curl	curl_fuzzer_http
freetype2	ftfuzzer
harfbuzz	hb-shape-fuzzer
jsoncpp	jsoncpp_fuzzer
lcms	cms_transform_fuzzer
libjpeg-turbo	libjpeg_turbo_fuzzer
libpcap	fuzz_both
libpng	libpng_read_fuzzer
libxml2	xml
mbedtls	fuzz_dtlsclient
openssl	x509
openthread	ot-ip6-send-fuzzer
php	php-fuzz-parser
proj4	proj_crs_to_crs_fuzzer
re2	re2_fuzzer
sqlite3	ossfuzz
systemd	fuzz-link-parser
vorbis	decode_fuzzer
woff2	convert_woff2ttf_fuzzer
zlib	zlib_uncompress_fuzzer
arrow	parquet-arrow-fuzz
aspell	aspell_fuzzer
assimp	assimp_fuzzer
ffmpeg	ffmpeg_demuxer_fuzzer
file	magic_fuzzer
grok	grk_decompress_fuzzer
libaom	av1_dec_fuzzer

Table 3.2. FuzzBench and SBFT'23 benchmarks tested and fuzzing binaries analyzed

3.3.2 Testing with FuzzBench

FuzzBench projects were built and tested almost identically to the methodology shown for OSS-Fuzz (see 3.2.2), except for a few small differences.

Initially, I used the “helper.py” script to download the latest Docker image for each project, but the fuzzers had to be built manually: this is because all FuzzBench benchmarks are compiled using a custom combination of sanitize flags from ASan and UBSan [14], implying that FuzzBench does not perform memory analysis when fuzzing. To replicate this build configuration, I had to modify the environment variables provided by the Docker images, using the following command:

```
$ docker run --rm --privileged
    --platform linux/amd64 --memory=16g
    -e PROJECT_NAME={project_name} -e HELPER=True
    -e FUZZING_LANGUAGE=c++ -e FUZZING_ENGINE=afl
    -e SANITIZER=address
    -e SANITIZER_FLAGS_address="
        -fsanitize=address,array-bounds,bool,builtin,enum,
            integer-divide-by-zero,null,object-size,return,
            returns-nonnull-attribute,shift,
            signed-integer-overflow,
            unsigned-integer-overflow,
            unreachable,vla-bound,vptr
        -fno-sanitize-recover=array-bounds,bool,builtin,enum,
            integer-divide-by-zero,null,object-size,return,
            returns-nonnull-attribute,shift,
            signed-integer-overflow,unreachable,
            vla-bound,vptr
        -fsanitize-address-use-after-scope"
    -v /oss-fuzz/build/out/{project_name}/:/out/
    -v /oss-fuzz/build/work/{project_name}/:/work/
    -v /oss-fuzz/build/corpus/{project_name}/:/corpus/
    -v /home/zio-saba/Scrivania/TESI/logfiles/:/logfiles/
    -it  gcr.io/oss-fuzz/{project_name} /bin/bash
```

As previously shown, the first parameters are needed to create a privileged instance of Docker, specify the running platform on which the fuzz targets will be tested and limit memory usage, the arguments starting with `-e` override the environment variables provided by the Docker image, the ones starting with `-v` are used to create shared directories and the last line invokes the project image to load as well as making it interactive by spawning a `/bin/bash` process. Given that the container accepts only one sanitizer at a time, I had to specify both ASan and UBSan flags in the same variable storing the ASan sanitizer flags, using as reference the subset of UBSan flags used by default by FuzzBench [14].

All crashes collected during the initial analysis were then categorized as follows:

- **crash:** inputs leading to any scenario that forces the system to close the process, like SEGV and buffer overflows
- **out-of-memory (oom):** inputs inducing memory leaks or using huge allocation sizes to purposely test fail-safe mechanisms
- **timeout:** inputs that are either very long to parse or that purposely introduce unnecessary operations, again to test the resiliency of the program

Finally, tests were performed using the following command:

```
$ for i in /corpus/*; do
    echo "TEST" $i;
    echo "TEST" $i >> /logfiles/PROJECT_SANITIZER-NAME.log;
    timeout 30s ./{fuzz_target} $i
        &>> /logfiles/PROJECT_SANITIZER-NAME.log;
  done
```

As before, the shell `for` construct scans all files found in the "corpus" directory, then prints the name of the current testcase on the terminal for monitoring purposes, and finally logs the name of the current testcase and the results of the fuzz target executed on that particular testcase on a log file. The only addition is the `timeout` command, and that is because due to the presence of inputs that purposely take unnecessary time, I decided to follow the same timeout duration used by ClusterFuzz when fuzzing with AFL++ [9].

3.4 Bug Deduplication

To correctly analyze and categorize the bugs discovered, it is necessary to perform *bug deduplication*, a technique used to identify all inputs that produced the same output, which is crucial to remove unnecessary data from the results and make the bug analysis work less tedious for the developer. There are several strategies that can be applied to deduplicate bugs, but it is important to mention that there is no fool-proof methodology that produces perfect results on all possible scenarios, and this is because which information you are using as reference and their number will (almost always) lead to partial loss of data.

We provide an overview of the most popular bug deduplication techniques [49]. The most common technique is *stack trace analysis*, i.e. analyzing input ID, function call records (also called *frames*) and error type: this is because when a program crashes, several debug and error information are collected from the runtime stack, and their analysis has proven to be particularly effective in identifying the uniqueness of a bug. Another approach is based on *code coverage information analysis*: in the context of fuzzing, the fuzzing tool maintains several bitmaps to store the execution path of each testcase, so that when a bug is encountered, it is considered unique only if the testcase follows a previously unseen execution path or if it matches an already known flow but with a different execution order. Finally, *context comparison* may be used to differentiate between different execution scenarios: most common methodologies are taint analysis, which involves analyzing how data flows and is manipulated by the program during its execution, and symbolic execution, which performs a static analysis of the code to simulate all possible execution scenarios. However, this last approach is often discarded, as it requires enormous amounts of time, data and knowledge to produce meaningful results.

Since this work focused primarily on frameworks based on ClusterFuzz as the back-end, I decided to employ the same *stack trace analysis* approach used by ClusterFuzz when generating bug reports, where the fields “Crash Type” and “Crash state” are the main references (see Figure 3.2):

- address bugs were deduplicated using as reference the last 3 stack entries and the error type
- undefined behavior bugs were deduplicated using as reference the error type, the address where the error occurred and the causing instruction
- memory bugs were deduplicated using as reference the last 3 stack entries, the error type and (if available) the signal that caused the spontaneous crash

Chapter 4

Results

All collected bugs were appropriately deduplicated: to do this, I used several Python scripts (one for each sanitizer and one for Valgrind) that analyzed the logs generated during each fuzzing session, performing deduplication according to their respective methodology described in 3.4.

Then, *bug triage* was performed, analyzing more in-depth where the bug occurred, what happened and why: this required a manual investigation of each bug individually, statically and dynamically, to have a clearer understanding of the problem and potentially provide suggestions to the developers regarding the fix. This step was also important to refine the previous deduplication step, as two inputs that triggered (apparently) different bugs may only have distinct stack flows: if the bug originated from the same function, it meant that two different execution flows triggered the same error. In this case, a common practice is to fix one of them and then use the other input as an additional check for correctness.

Finally, all recorded bugs for a project, along with their log and fuzz targets, were reported to their respective developers according to their preferred communication method: almost all reports were sent using GitHub's integrated issue tracker, some required a subscription to a proprietary issue tracking platform, and a few of them were sent via emails. When available, an estimated priority was assigned to the overall content of the report, usually with the values “Low”, “Medium” and “High”.

At the moment of writing, unfortunately, not all developers answered and/or acknowledged the reported bugs.

4.1 OSS-Fuzz

Project	Sanitizers	Queue size	Crashes	ASan	Valgrind	UBSan	Total	Confirmed
binutils	ALL	20, 274	0	0	1	0	1	1
harfbuzz	ALL	23, 357	0	0	1	0	1	1
imagemagick	ALL	9, 470	0	0	6	1	7	1
libxml2	ALL	13, 474	0	0	0	0	0	0
skia	ALL	18, 295	0	0	0	0	0	0
gpsd	A+M	5404	0	0	0	0	0	0
libyang	A+M	8, 745	0	0	0	0	0	0
llvm	A+M	10, 657	0	21	0	0	21	16
openjpeg	A+M	8, 856	0	0	0	1	1	0
wasmedge	A+M	9, 454	0	0	0	0	0	0
cairo	A+UB	15, 870	0	1	1	28	30	0
clamav	A+UB	6, 742	0	0	0	2	2	0
freerdp	A+UB	7, 607	0	0	0	1	1	1
tarantool	A+UB	10, 987	0	0	1	0	1	1
vlc	A+UB	16, 018	2	1	2	4	9	5
fwupd	ASan only	5, 843	0	0	0	0	0	0
glslang	ASan only	14, 534	0	1	1	0	1	1
inchi	ASan only	12, 034	0	1	4	3	8	8
radare2	ASan only	9, 914	0	1	0	9	10	10
zeek	ASan only	8, 390	0	0	1	5	6	6
fluent-bit	NONE	4, 968	0	0	1	1	2	2
gpac	NONE	22, 917	2	0	25	0	27	27
libdwarf	NONE	7, 667	0	0	0	0	0	0
libredwg	NONE	46, 160	0	1	3	0	4	4
serenity	NONE	9, 940	0	0	1	1	2	2
TOTAL BUGS			4	27	48	56	134	86

Table 4.1. Bugs collected for OSS-Fuzz projects

Table 4.1 shows the aggregated results for OSS-Fuzz, dividing bugs per category and showing the total number of bugs found compared to the number of bugs that were confirmed by the respective developers after being reported. While it is important to note that this is a very small subset of projects tested (25 out of 1000+) and is not representative of the entire OSS-Fuzz campaign, it is still a good indicator that there are indeed bugs that these frameworks are missing.

This analysis found a total of 134 bugs (an average of 5.36 bugs per project). In total, 63 bugs out of 134 were found by enabling new sanitizers for the first time in a project: this result is expected since that software was never tested for that class of bugs, and doing it for the first time will surely detect new bugs. On the other hand, the remaining 71 bugs were found using sanitizers that were already enabled in the target software: this result is more interesting because it means that even if a sanitizer is enabled, OSS-Fuzz may detect a bug but fails to report it.

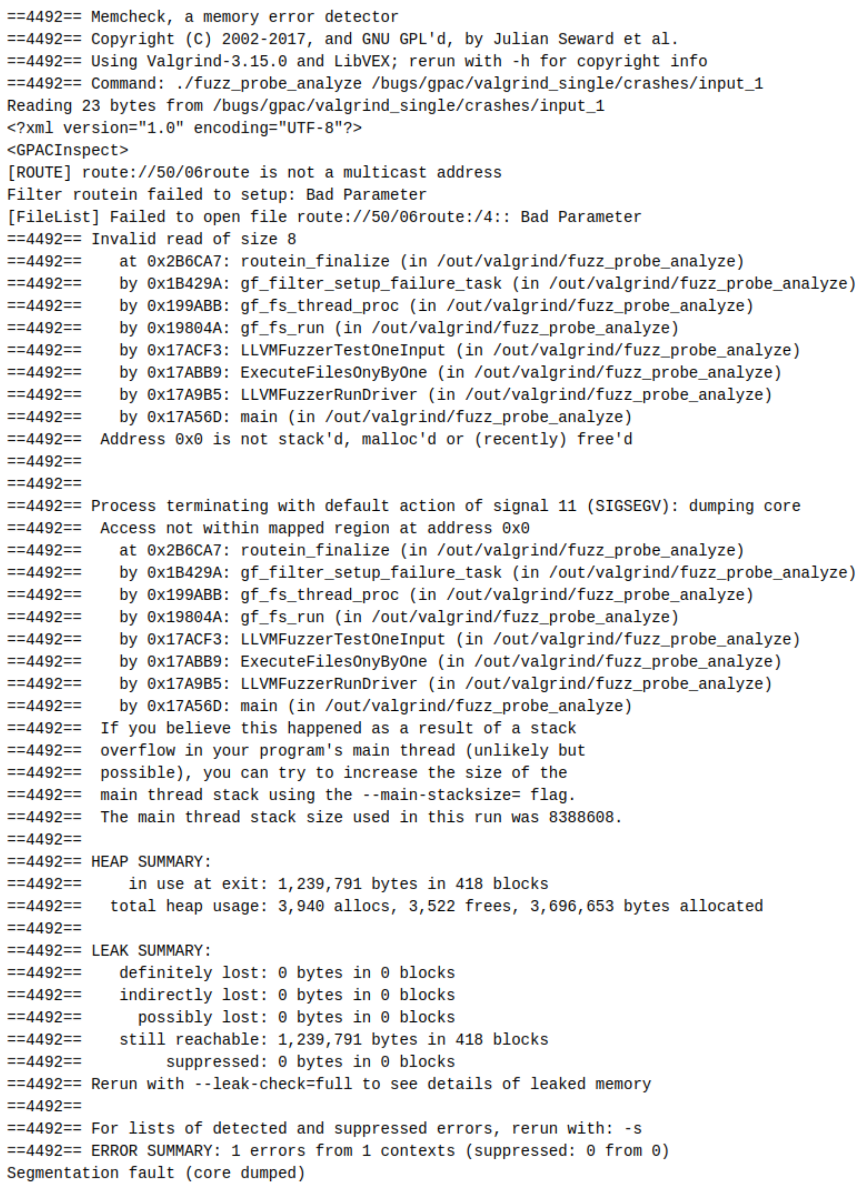
The “LLVM” project yielded a total of 21 bugs, mostly stack exhaustion bugs which cannot reproduce outside of builds with ASan enabled. This is a known issue with ASan, as it increases the size of stack objects thus exhausting the stack faster than the program would without the sanitizer: we further confirmed that, by doubling the maximum size of the stack, even the ASan build could not exhaust the stack. We still reported these issues and deferred to the developers the choice on whether these issues are relevant, and the discussion is still ongoing.

We can also infer the popularity of each sanitizer. ASan is the most widely used, which is reflected by the number of bugs it found. UBSan is also among the most used ones, although the statistic shown here is biased by “Cairo”: while this project is affected by many UB bugs, specifically the improper conversion of data between different types, almost all previous reports on the issue tracker related to similar problems were simply marked as “WontFix”. Finally, almost all projects had (at least) 1 memory bug discovered using Valgrind, which shows how MSan is often discarded due to its reports having too many false positives when the program is not properly compiled with this sanitizer.

4.1.1 Case Study: GPAC

The *"GPAC Project on Advanced Content"* (abbreviated *GPAC*) [31] is a free and open-source multimedia framework that provides tools to process, inspect, package, stream, media playback and interact with media content, making it a popular choice also among major broadcasters such as Netflix.

Its analysis yielded 27 bugs, with 2 crashes and 25 memory-related bugs. The crashes were related to a SEGV signal sent as a consequence of the process attempting to access address `0x0` (i.e. zero page), which is the first page of a computer's memory and whose access is prohibited and causes an access violation fault, shown below:



```
==4492== Memcheck, a memory error detector
==4492== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4492== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4492== Command: ./fuzz_probe_analyze /bugs/gpac/valgrind_single/crashes/input_1
Reading 23 bytes from /bugs/gpac/valgrind_single/crashes/input_1
<?xml version="1.0" encoding="UTF-8"?>
<GPACInspect>
[ROUTE] route://50/06route is not a multicast address
Filter routein failed to setup: Bad Parameter
[FileList] Failed to open file route://50/06route:/4:: Bad Parameter
==4492== Invalid read of size 8
==4492==    at 0x2B6CA7: routein_finalize (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x1B429A: gf_filter_setup_failure_task (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x199ABB: gf_fs_thread_proc (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x19804A: gf_fs_run (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17ACF3: LLVMFuzzerTestOneInput (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17ABB9: ExecuteFilesOnyByOne (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17A9B5: LLVMFuzzerRunDriver (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17A56D: main (in /out/valgrind/fuzz_probe_analyze)
==4492==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==4492==
==4492==
==4492== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==4492==  Access not within mapped region at address 0x0
==4492==    at 0x2B6CA7: routein_finalize (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x1B429A: gf_filter_setup_failure_task (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x199ABB: gf_fs_thread_proc (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x19804A: gf_fs_run (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17ACF3: LLVMFuzzerTestOneInput (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17ABB9: ExecuteFilesOnyByOne (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17A9B5: LLVMFuzzerRunDriver (in /out/valgrind/fuzz_probe_analyze)
==4492==    by 0x17A56D: main (in /out/valgrind/fuzz_probe_analyze)
==4492==  If you believe this happened as a result of a stack
==4492==  overflow in your program's main thread (unlikely but
==4492==  possible), you can try to increase the size of the
==4492==  main thread stack using the --main-stacksize= flag.
==4492==  The main thread stack size used in this run was 8388608.
==4492==
==4492== HEAP SUMMARY:
==4492==     in use at exit: 1,239,791 bytes in 418 blocks
==4492==   total heap usage: 3,940 allocs, 3,522 frees, 3,696,653 bytes allocated
==4492==
==4492== LEAK SUMMARY:
==4492==    definitely lost: 0 bytes in 0 blocks
==4492==    indirectly lost: 0 bytes in 0 blocks
==4492==      possibly lost: 0 bytes in 0 blocks
==4492==    still reachable: 1,239,791 bytes in 418 blocks
==4492==         suppressed: 0 bytes in 0 blocks
==4492== Rerun with --leak-check=full to see details of leaked memory
==4492==
==4492== For lists of detected and suppressed errors, rerun with: -s
==4492== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Figure 4.1. Memory access violation reported by Valgrind

It is interesting to note that these bugs were already reported by ClusterFuzz when discovered by this work and, more importantly, they were still in the bug disclosure windows: this meant that the inputs causing these bugs were not meant to be added to later fuzzing queues and should have not been publicly accessible and available, due to obvious security implications.

The remaining 25 memory bugs were all related to use-of-uninitialized-memory (UUM) and memory-bound errors. More specifically, they can be divided into three categories according to Valgrind:

- *conditional jump or move depends on uninitialized value*: happens when an `if` statement is based on a variable whose value is uninitialized
- *use of uninitialized value in function F*: happens when one (or more) of the parameters passed to function F are uninitialized
- *invalid read of size N*: happens when a read/write operation performed overruns the provided buffer of N bytes, as the content outside of a buffer could be anything

After reporting the UUM bugs and having the pleasure of interacting with the CEO of GPAC itself, we were happy to discover all bugs fixed in just a few days. The developers created a public commit [28] to explain the issues discovered and reassure everyone that they did not pose a security concern, as they believe the path to exploitation is difficult. However, they agreed on the fact that such bugs could be resolved by trivial and obvious corrections, and the existence of such issues shows that they need to be more vigilant: most of the problems mentioned above originated from local structures declared but left uninitialized, buffer-related operations with no checks regarding the size of the data passed to the buffer, and recalling that most read/write functions return the number of bytes actually read/written. They concluded by expressing their gratitude, as they have recently been battling with seemingly random bugs causing disruptions with the normal functionality of the program, therefore the report has been valuable in helping them to remove uninitialized variables that were leading to undefined behavior.

This project was a clear example of the importance of using sanitizers during tests, as they highlighted simple and trivial errors that would otherwise be easily overlooked during a manual analysis of the code, but still that greatly helped them in improving the user experience by removing unwanted behavior.

4.1.2 Case Study: VLC

The *"VLC Media Player"* (commonly known as *VLC*) [57] is a free and open-source media player software and streaming media server developed by the "VideoLAN Project", supporting many audio and video compression methods, file formats and providing many free decoding and encoding libraries.

Its analysis yielded a total of 9 bugs spread across all categories: 2 crashes, 1 heap-buffer overflow, 2 memory-related bugs and 4 undefined-behaviors. The heap-buffer overflow bug was initially presented as an ASan bug related to an invocation of `memcpy` that was copying a structure into another one of the same type, which initially left the developers a bit perplexed. After further analysis, they discovered that the input provided was triggering some strange behavior in another function of their program, whose output led to incorrect usage of the copy function and therefore the buffer overflow. The crashes were both related to a SEGV signal sent as a consequence of the process attempting to access a low address that was not mapped within the process's memory region, causing an access violation fault: both problems originated from a series of read/write operations performed on uninitialized values causing undefined behavior, ultimately leading to the process accessing an erroneous address. The first memory bug was related to a memory-bound error caused by several read operations overrunning the provided buffers, leading to undefined behavior. At the moment of writing, the problem has yet to be fixed because the developers are discussing whether to rewrite or remove the source file causing the problem, as they argue that it is very insecure and could pose a threat to the overall security of the product. The second memory bug was related to a use-of-uninitialized-memory (UUM) bug when opening a provided input media file, but the proposed fix has yet to be merged with the main code. Finally, the undefined-behavior (UB) bugs were all related to signed integer-overflow errors, which are relatively easy to fix. Yet, at the moment of writing, they have not been acknowledged by the developers.

This project highlighted the most common behavior encountered during this work when interacting with open-source developers. When presented with bugs that could potentially lead to vulnerabilities or problems in the normal functionalities of their program, their response is usually quick and the problem is fixed in a short amount of time. Instead, when the problems have low or negligible priority (like the undefined-behavior bugs), they either ignore the reports or acknowledge their existence and postpone them indefinitely.

4.2 FuzzBench

Project	Sanitizers	Queue size	Crashes	ASan	UBSan	Total	Confirmed
bloaty	NONE	2349	0	0	2	2	2
curl	NONE	0	0	0	0	0	0
freetype2	ALL	0	0	0	0	0	0
harfbuzz	ALL	34634	0	0	0	0	0
jsoncpp	A+UB	0	0	0	0	0	0
lcms	ALL	30526	0	0	0	0	0
libjpeg-turbo	ALL	0	0	0	0	0	0
libpcap	ALL	111	0	0	0	0	0
libpng	ALL	0	0	0	0	0	0
libxml2	ALL	1961857	0	0	0	0	0
mbedtls	NONE	1	0	0	0	0	0
openssl	A+UB	0	0	0	0	0	0
openthread	A+UB	0	0	0	0	0	0
php	ALL	202235	0	0	0	0	0
proj4	NONE	208971	0	0	0	0	0
re2	ALL	0	0	0	0	0	0
sqlite3	ALL	0	0	0	0	0	0
systemd	ALL	238072	0	0	0	0	0
vorbis	A+M	0	0	0	0	0	0
woff2	ALL	0	0	0	0	0	0
zlib	ALL	0	0	0	0	0	0
arrow	NONE	1163787	0	0	0	0	0
aspell	A+UB	919701	0	0	2	2	2
assimp	A+UB	312651	43	104	13	160	64
ffmpeg	ALL	212617	0	0	6	6	0
file	A+UB	1916360	0	0	0	0	0
grok	ALL	1492374	0	2	15	17	17
libaom	ALL	19729	0	0	0	0	0
TOTAL BUGS			43	106	38	187	85

Table 4.2. Bugs collected for Fuzzbench and SBFT'23 benchmarks

Table 4.2 shows the aggregated results for FuzzBench, dividing bugs per category and showing the total number of bugs found compared to the number of bugs that were confirmed by the respective developers after being reported. For the sake of completeness, the table also distinguishes between the standard FuzzBench benchmarks (above) and the "SBFT '23" benchmarks (below), and reports the sanitizers regularly used by the OSS-Fuzz counterpart of the selected projects.

It is immediate to see that all bugs came from the "SBFT '23" benchmarks: these results were unexpected, given that these programs have been already tested extensively during the tool competition hosted by this conference, especially "ASSIMP" which is responsible for the majority of them. These results show the importance of double-checking the possible presence of a bug found in an older version of a software in the latest version as well. The bugs we found were detected more than a year ago but left unreported until now: an attacker could have simply downloaded data from the SBFT '23 tests and gained access to security-related bugs with very little effort.

4.2.1 Case Study: ASSIMP

The *"Open Asset Import Library"* (abbreviated *ASSIMP*) [27] is a cross-platform library to import 3D models into a shared, in-memory immediate format, also providing a common API for different 3D asset file formats.

Its analysis yielded a total of 160 bugs, 96 of which were unreported, divided as follows: 1 stack-buffer-overflow, 1 stack-overflow, 71 heap-buffer-overflows, 20 crashes and 3 undefined behaviors. The stack-buffer-overflow error was related to a `memcpy` invocation not properly checking the length of the involved buffers, eventually overriding the stack and causing a segmentation fault. The stack-overflow was triggered by an `open` function triggered in a never-ending loop, eventually leading to stack exhaustion. The heap-buffer-overflows and crashes were mostly related to improper sanitizations of user inputs when performing memory accesses, eventually also leading to segmentation faults. All the aforementioned scenarios are some of the most common methodologies that malicious people use when exploiting a program to gain control of your system, therefore fixing them would be crucial.

We reported all the bugs we found with a (private) security report through the GitHub platform this project uses. At the time of writing, the developers have not yet publicly acknowledged the existence of those bugs, but they have started providing fixes to them. Given the high number of reported bugs, it is reasonable to expect the bug-fixing process to take some time, but it will eventually lead to a much safer and, hopefully, bug-free library.

4.3 Developers' Responses

This study has shown that developers have different approaches to bug reporting.

The OSS-Fuzz reports were submitted during September and, at the time of writing, almost one third of the bugs discovered have yet to be acknowledged by their respective developers. As previously shown, most of them were related to use-of-uninitialized-memory and undefined behaviors, but the reports received mixed responses from the developers. The majority either answered directly after the problem was fixed, or acknowledged the correctness of the bugs and the content of the report while promising to fix them in future versions, a suboptimal approach that shows how many open-source developers either do not have enough time and resources to fix them, or they simply do not care about reports on trivial bugs that are not security relevant and postpone them (sometimes indefinitely). There were some cases where the reports were accepted but with negative interactions, as the developers acknowledged the bugs but reprimanded the submitter: this was either because the work presented was effectively a duplicate of OSS-Fuzz and therefore deemed useless (which indeed it was not), or because they thought the person who wrote them wanted to "collect yet another bug report" at their expense. Unfortunately, this highlights a bad practice when it comes to bug reporting: some developers expect the person who reports a bug to also provide a complete analysis and/or fix to lessen their work, which would obviously require extensive skills and knowledge of the project codebase, while some users send general or incomplete bug reports simply for the sake of adding a new bug under their name, which many developers consider to be harmful to their popularity because a buggy program will eventually become less trusted. Finally, there were also a few cases where the submitted reports were rejected or simply ignored altogether. For example, when presented with a series of UUM bugs found by Valgrind, one developer replied by stating that configuring the tool to suppress a specific set of (apparently) false-positive errors from a specific list removed all the errors reported. In another case, the report was rejected because the developer did not trust the origin of the resources provided (i.e. buggy testcase and binary tested) for his own security. There was also an instance where the developer rejected the report because the provided testcase was too much specifically crafted to be a corner-case, in his opinion not effectively a real scenario that could pose a threat and therefore will not be fixed.

The FuzzBench reports were submitted during December, and at the time of writing all the bugs except the "ASSIMP" ones have been confirmed. As previously mentioned, we found a non-trivial amount of bugs in this project, promptly notified in a secure manner while also suggesting a "High" priority for the related bug report. To help the developers, the report also categorized and grouped all the bugs. We noticed that bugs are being fixed, even if no official answer has been made to our report yet.

Perceived priority. How developers perceive a bug report's priority varies greatly, and this difference becomes even more apparent when considering different kinds of bugs. In general, most bugs reported using ASan have been confirmed (at most) within 2 weeks of being reported, and then triaged and fixed shortly after. Using memory addresses incorrectly in a program will most likely cause problems on the stack and heap, easily leading into heap-based and stack-based corruption, and both scenarios are known to be some of the most popular attack vectors exploited by malicious people. Similarly, also most memory-related bugs reported using Valgrind have been confirmed and fixed in a matter of weeks. However, almost all bugs reported using UBSan have been ignored, with few exceptions: this is because undefined-behavior bugs often originate from basic arithmetic and logic operations, with a common belief that such mistakes are unlikely to be too harmful and can be addressed later. A simple search on the "Common Vulnerabilities and Exposures" (CVE) database [43] combining the keyword "integer overflow" with common attacks like "privilege escalation" and "remote command execution" produced thousands of results, showing how even the most basic, trivial and simple mistakes could have catastrophic consequences.

Overall, most reports produced were accepted and fixed, but this work highlighted how open-source developers tend to focus on bugs that could pose a real threat to their product, rather than polishing and fixing small bugs that could still lead to unexpected behavior or unwanted interactions, ultimately leading to a worse user experience. Moreover, as discussed earlier, there is a margin of distrust between developers and users, which should be one of the key factors in the development of open-source projects (recall 2.3), but instead has gained a negative reputation due to bad habits and perhaps too many expectations from both sides.

4.4 Discussion

This section contains a discussion of the results from the autonomous fuzzing infrastructures evaluated in this thesis, i.e. OSS-Fuzz and FuzzBench.

OSS-Fuzz. Albeit obtained from a small subset of projects that are not representative of the entire campaign, the results obtained show that this framework is missing a relevant number of bugs. Furthermore, although not shown in Table 4.1, less than half of the projects tested included AFL++ as one of the fuzzing engines used for the tests: it is interesting to note that all the projects that used AFL++ yielded (on average) less bugs than the others, which again highlights why it is considered the current state-of-the-art fuzzer.

The OSS-Fuzz results were indeed unexpected as one would assume that relying on this infrastructure, especially given the organization running it and the scale of such campaign, would be an effective strategy to ensure that your program is tested correctly and efficiently. The reasons why this work discovered so many bugs are up for debate, as the most inner workings of OSS-Fuzz and its back-end ClusterFuzz are not publicly available in their entirety, probably for security reasons; however, it is apparent that design and implementation choices as well as open-source developers involvement are all factors that contributed to the following key hypothesis.

A trivial explanation lies in the developers' choices when it comes to selecting which sanitizers and fuzzing engines will be used for the tests. Regarding the sanitizers, one would expect to find many bugs of a certain type when introducing a sanitizer that is not already being used by OSS-Fuzz. Also fuzzing engines, similarly to sanitizers, use different approaches and strategies when testing a program: given that each fuzzer generates different new testcases and produces different coverage, using the same corpus with a different combination of fuzzer and sanitizers may produce substantially different results.

Another reason may lie in the algorithms used by ClusterFuzz to manage the fuzzing queues. Each fuzzing session produces as output a corpus containing all the "interesting input" discovered (i.e. that produced new coverage) along with any new testcases generated by the fuzzer, which is then manipulated and processed to form what will be the corpus used as input for future fuzzing sessions. Among the many operations performed, *bug deduplication* and *pruning* are essential to make sure that the size of this corpus does not explode over time. We already established how ClusterFuzz performs *deduplication* (i.e. "error type" and the last 3 stack entries), and also mentioned that there is no general solution to this process that does not involve some degree of information loss. The same can be said about *pruning*, which is the process of removing all unnecessary and non-relevant inputs from a corpus. ClusterFuzz employs several *"Multi-Armed Bandit (MAB)"* [60] algorithms, whose statistical explanation will be omitted for the sake of simplicity, in the following way: at the end of each fuzzing session, ClusterFuzz must decide whether to prioritize inputs that caused bugs or inputs that produced new coverage, using regress knowledge and implementation choices as reference. Given that ClusterFuzz's developers prioritize code coverage, the resulting fuzzing queue will contain mostly that type of inputs, implying that when pruning is performed it will also most likely remove some inputs causing bugs from future fuzzing queues.

Related to the problem of pruning, there is also the versioning problem. Let us assume that it exists a specific input on the current version of the program that causes a bug. Then, the developers release a new version introducing some changes, including a fix for said bug, and the input is automatically tested (successfully) and removed by ClusterFuzz as it is not relevant anymore. Patching a single input does not necessarily mean that particular type of error has been permanently fixed: given that the input has been removed from the queue, unless another input on future fuzzing queues reproduces the problem, it persists. This is the reason why keeping old inputs that caused bugs and testing old corpora on newer versions is useful, as it is not uncommon for developers to reintroduce old bugs during changes, and they might produce interesting and unexpected results.

Finally, we mention how ClusterFuzz handles bugs that cannot be reliably reproduced [23]: *"ClusterFuzz does not consider testcases that do not reliably reproduce as important. However, if a crash state is seen very frequently despite not having a single reliable testcase for it, ClusterFuzz will file a bug for it. When ClusterFuzz finds a reliably reproducible testcase for the same crash state, it creates a new report and deletes the older report with the unreliable testcase."* During this analysis, there were a few inputs that did not produce a bug consistently (i.e. the bug did not appear on all executions with the same testcase as input), something that was obviously mentioned in the reports containing them. Given that it is unknown how many times a crash must be "seen" by ClusterFuzz before a generic bug report for it is produced, it is possible that the reports generated during this work may be related to bugs already known by ClusterFuzz but that were not appearing enough times to be considered relevant.

Overall, the results shown in Table 4.1 proved the effectiveness of combining the use of multiple sanitizers (or at least as many as possible) with different fuzzing engines so that there are no missed bugs during the continuous fuzzing process proposed by OSS-Fuzz. In addition, the maintainer of this framework should periodically review how fuzzing queues are managed and tested, to ensure that there are no issues in their testing pipelines and that no potential bugs are kept publicly available by users downloading such queues. Finally, the project developers should also periodically check the correctness of the settings used for the tests, to ensure that all tests are always performed to be as effective and accurate as possible.

FuzzBench. The standard FuzzBench benchmarks and the additional benchmarks from the "SBFT '23" conference were tested using their OSS-Fuzz counterparts, as we have already mentioned that FuzzBench performs tests on older versions that are known to contain bugs and using the same set of input corpora across different executions for reference purposes. Even when tested with OSS-Fuzz on the latest version, all default benchmarks did not produce any new bugs, which is certainly an unexpected but welcome result as it shows the effectiveness of continuous and rigorous testing. In fact, although not shown in Table 4.2, almost all the projects tested used all the available fuzzing engines provided by ClusterFuzz (i.e. LibFuzzer, AFL++ and Honggfuzz), which shows how *ensemble fuzzing* provides overall better results when different fuzzers, sanitizers and resulting testcases are efficiently mixed together. Instead, all bugs discovered with FuzzBench came from the set of projects used in the “SBFT ’23” conference, which came as a surprise as one would think that being them the object of a tool competition also implied that they were thoroughly tested, revised and fixed after the competition ended. This is especially true for the “ASSIMP” project, as we argue it is unacceptable for a project that has managed to satisfy the rigid OSS-Fuzz requirements of being a highly trusted and regarded project by the entire open-source community, to also produce so many bugs and crashes.

In general, when testing a program, it is good practice to periodically check if older bugs are still present in the latest version, as it is not uncommon for developers to make changes to the code that mistakenly reintroduce a previously fixed bug. This statement should apply to any automated fuzzing framework, such as those evaluated in this thesis, as well as events and conferences like the “SBFT ’23”, where those responsible for the fuzzing tool challenge failed to do so and left many valuable bugs open to anyone but unreported.

Chapter 5

Conclusions and Future Work

In this last chapter, we will summarize the work presented in this thesis. After that, we will present some directions for future work that could be explored to further improve our solution and the impact it can have in malware analysis research.

5.1 Conclusion

We first provided the concepts of fuzzing and sanitizers, a popular bug-detection approach in testing environments that has proven to be particularly effective, especially when combined with the "Continuous-Fuzzing/Continuous-Integration" pipeline that many modern organizations employ when developing their products. Then, we introduced autonomous fuzzing frameworks, how they work and described the chosen frameworks analyzed. Followed a description of the methodology used to analyze said frameworks along with how bugs were collected, analyzed and reported. Concludes a thorough analysis of the results, discussing how developers perceived the bug reports created and the implications of the behaviors observed with some case studies.

This study highlighted the popularity of autonomous fuzzing infrastructures and their effectiveness, but also that the lack of standardized approaches and design trade-offs are major contributors to their accuracy when it comes to automatically detecting and reporting bugs. Although the overall number of projects analyzed is not representative of the entire campaigns, the applied methodology still managed to discover several hundreds of previously unreported bugs, showing that the workflow adopted by the analyzed campaigns presents flaws that lead to overlooked bugs and potential vulnerabilities.

In today's world, where software development techniques are constantly changing and improving, it is crucial to ensure that all aspects of software development progress at the same pace, including the "Testing" phase: many organizations rely on automated testing, a time-efficient and effective solution, but that comes with its shortcomings as highlighted by this work. Therefore, while relying on autonomous testing techniques and infrastructures has proven its efficacy in time, developers should monitor, configure and employ these tools regularly and keep them up to date to ensure that also the testing environments can be as accurate and productive as possible.

5.2 Future Work

This work highlighted relevant issues in the way fuzzing infrastructures detect bugs and vulnerabilities, providing both practical and research contributions. Still, we strongly believe this could be the starting point for more research in the field of autonomous fuzzing infrastructures. In particular, we foresee two main directions in which this thesis can be expanded.

We have shown how two of the most popular fuzzing infrastructures have issues in accurately detecting some bugs, leaving them accessible to malicious entities that can potentially use them to attack digital systems. While OSS-Fuzz and FuzzBench are two prominent examples of such infrastructures, more fuzzing infrastructures exist and are created as time passes: we believe expanding this analysis to more infrastructures would provide valuable information on how to design more accurate fuzzing infrastructures. Doing so would not only provide the practical value of finding overlooked bugs in specific environments, but could also provide new insights for other infrastructures to improve. For instance, the analysis of OSS-Fuzz showed the importance of sanitizers; FuzzBench, on the other hand, showed us the importance of always testing bugs in the latest version of the software as well as the ones used for testing. The relevance of these two pieces of information goes far beyond the boundaries of the aforementioned infrastructures, and we believe analyzing more of them will provide even more valuable information.

Other than testing more infrastructures, we believe this work by itself could be the starting point to create a set of guidelines on how to design accurate fuzzing infrastructures. We highlighted some critical points that are relevant to fuzzing in general, like the importance of sanitizers, the importance of testing the latest versions, and the danger of sharing too much information without proper checking. We believe all these insights could be aggregated to create a common framework to allow future development of fuzzing infrastructures to be as complete as possible, hopefully avoiding these mistakes in the future.

Bibliography

[1] AFL++. Fuzzing library “libafl”. https://aflplus.plus/libafl-book/libafl.html.

[2] Bashari Rad, B., Bhatti, H., and Ahmadi, M. An introduction to docker and analysis of its performance. *IJCSNS International Journal of Computer Science and Network Security*, (2017).

[3] Borzacchiello, L., Coppa, E., and Demetrescu, C. Fuzzolic: Mixing fuzzing and concolic execution. *Computers & Security*, (2021), 102368. Available from: https://www.sciencedirect.com/science/article/pii/S0167404821001929, doi:https://doi.org/10.1016/j.cose.2021.102368.

[4] Exploit Database. The shellshock incident explained. https://www.exploit-db.com/docs/english/48112-the-shellshock-attack-%5Bpaper%5D.pdf?ref=benheater.com.

[5] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association (2020). Available from: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[6] Forbes. The price of software errors. https://www.forbes.com/councils/forbestechcouncil/2023/12/26/costly-code-the-price-of-software-errors/.

[7] Foundation, P. S. Python 3.10.12 documentation. https://docs.python.org/release/3.10.12/.

[8] GitLab. Coverage-guided fuzz testing. https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/.

[9] Google. Clusterfuzz afl timeout. https://github.com/google/clusterfuzz/blob/ca619ef34aeaf35abf9e7bbcfd48b8f497e07ba3/src/local/butler/scripts/setup.py#L41.

[10] Google. Clusterfuzz project documentation. https://google.github.io/clusterfuzz/.

[11] Google. Clusterfuzzlite. https://google.github.io/clusterfuzzlite/.

[12] Google. Clusterfuzzlite github repository. https://github.com/google/clusterfuzzlite/.

[13] Google. Experiment configuration file. https://github.com/google/fuzzbench/blob/master/service/experiment-requests.yaml.

[14] Google. Fuzzbench base image sanitizer flags. https://github.com/google/oss-fuzz/blob/81b41ad37a95577aa34ffa1f0711d467f897a619/infra/base-images/base-builder/Dockerfile#L74.

[15] Google. Fuzzbench benchmarks. https://google.github.io/fuzzbench/reference/benchmarks/.

[16] Google. Fuzzbench project documentation. https://google.github.io/fuzzbench/.

[17] Google. Google open source project. https://opensource.google/.

[18] Google. Google sanitizers' github repository. https://github.com/google/sanitizers.

[19] Google. GoogleDriver for Google Chrome. https://developer.chrome.com/docs/chromedriver?hl=it.

[20] Google. gsutil tool. https://cloud.google.com/storage/docs/gsutil.

[21] Google. Oss-fuzz issue tracker. https://issues.oss-fuzz.com/issues.

[22] Google. Oss-fuzz project documentation. https://google.github.io/oss-fuzz/.

[23] Google. "Unreliable crashes" excerpt from ClusterFuzz Documentation. https://google.github.io/clusterfuzz/using-clusterfuzz/workflows/fixing-a-bug/#unreliable-crashes.

[24] Google Project Zero. Bug disclosure guidelines. https://googleprojectzero.blogspot.com/2015/02/feedback-and-data-driven-updates-to.html (2015).

[25] Heartbleed Foundation. The heartbleed bug explained. https://heartbleed.com/.

[26] Hex Rays. Ida pro disassebler website. https://hex-rays.com/ida-pro.

[27] Kim Kulling. ASSIMP project. https://www.assimp.org/index.html.

[28] Kim Kulling, Aurelien David. GPAC Report: "fix a bunch of use-of-uninitialized-memory errors". https://github.com/gpac/gpac/pull/2965.

[29] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 2123–2138 (2018).

[30] Klooster, T., Turkmen, F., Broenink, G., Hove, R. T., and Böhme, M. Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in ci/cd pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pp. 25–32 (2023). doi:10.1109/SBFT59156.2023.00015.

[31] Le Feuvre, J., Concolato, C., and Moissinac, J.-C. Gpac: open source multimedia framework. In *Proceedings of the 15th ACM International Conference on Multimedia*, p. 1009–1012. Association for Computing Machinery, New York, NY, USA (2007). Available from: https://doi.org/10.1145/1291233.1291452.

[32] Lemieux, C. and Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 475–485 (2018).

[33] Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. Fuzzing: State of the art. *IEEE Trans. Reliab.*, **67** (2018), 1199. Available from: https://doi.org/10.1109/TR.2018.2834476.

[34] Liu, D., Metzman, J., Böhme, M., Chang, O., and Arya, A. Sbft tool competition 2023 – fuzzing track (2023). Available from: https://arxiv.org/abs/2304.10070, arXiv:2304.10070.

[35] LLVM. Address sanitizer documentation. https://clang.llvm.org/docs/AddressSanitizer.html.

[36] LLVM. Libfuzzer documentation. https://llvm.org/docs/LibFuzzer.html.

[37] LLVM. Memory sanitizer documentation. https://clang.llvm.org/docs/MemorySanitizer.html.

[38] LLVM. Undefined behavior sanitizer documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[39] Metzman, J., Szekeres, L., Simon, L., Sprabery, R., and Arya, A. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 1393–1403 (2021).

[40] Microsoft. Microsoft announces new project onefuzz framework, an open source developer tool to find and fix bugs at scale. https://www.microsoft.com/en-us/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/.

[41] Microsoft. Onefuzz github repository. https://github.com/microsoft/onefuzz.

[42] Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. *Commun. ACM*, (1990), 32–44. Available from: https://doi.org/10.1145/96267.96279.

[43] MITRE Corporation. CVE Database. https://cve.mitre.org/index.html.

[44] Nethercote, N. and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, p. 65–74. Association for Computing Machinery, New York, NY, USA (2007). Available from: https://doi.org/10.1145/1254810.1254820.

[45] Nethercote, N. and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, p. 89–100. Association for Computing Machinery, New York, NY, USA (2007). Available from: https://doi.org/10.1145/1250734.1250746.

[46] NIST NVD. National Vulnerabilities Database: Vulnerability Categories. https://nvd.nist.gov/vuln/categories.

[47] Open Source Initiative. Open-source license definition. https://opensource.org/osd.

[48] Open Source Security Foundation. Open-source project criticality score. https://github.com/ossf/criticality_score.

[49] Qian, C., Zhang, M., Nie, Y., Lu, S., and Cao, H. A survey on bug deduplication and triage methods from multiple points of view. *Applied Sciences*, **13** (2023), 8788.

[50] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, vol. 17, pp. 1–14 (2017).

[51] Serebryany, K. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC (2017).

[52] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pp. 309–318 (2012).

[53] Seward, J. and Nethercote, N. Using valgrind to detect undefined value errors with bit-precision. pp. 17–30 (2005).

[54] Stepanov, E. and Serebryany, K. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 46–55. IEEE (2015).

[55] Valgrind. Memcheck: a memory error detector. https://valgrind.org/docs/manual/mc-manual.html.

[56] Valgrind. Valgrind website. https://valgrind.org/.

[57] VideoLAN Organization. VideoLAN project. https://www.videolan.org/videolan/.

[58] Vijayasree, D., Roopa, N. S., Arun, A., et al. A review on the process of automated software testing. *arXiv preprint arXiv:2209.03069*, (2022).

[59] W3C. W3C WebDriver Documentation. https://w3c.github.io/webdriver/.

[60] Wikipedia. Multi-armed bandit algorithms. https://en.wikipedia.org/wiki/Multi-armed_bandit.

[61] Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 745–761. USENIX Association, Baltimore, MD (2018). ISBN 978-1-939133-04-5. Available from: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[62] Zalewski, M. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt (2014).

[63] Zalewski, M. Winafl github repository. https://github.com/googleprojectzero/winafl (2014).

[64] Zhen Yu Ding. Monorail scraper github repository. https://github.com/zhenyudg/monorail-scraper.