Universiteit Leiden

Master Computer Science

Automated Fuzzing for C/C++ Applications using Large Language Models and Control Flow Graph Analysis

Name:	Rajeck Massa
Student ID:	S2848457
Date:	28/07/2025
Specialisation:	Advanced Computing and Systems
1st supervisor:	Dr. ir. E. Makri
2nd supervisor:	Dr. O. Gadyatskaya

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Cyber security is more important than ever. Attacks on critical infrastructure are daily occurrences, raising the importance of secure software and patching found bugs in time. One method to find new bugs in software is fuzzing. Currently, it can be hard to implement fuzzing in the development phase due to requiring a lot of manual work and experience. In this work, we present a tool, called LLM2Fuzz, which can automate the fuzzing process of C-style applications by letting Large Language Models generate the fuzzing harness, custom mutators and seed input files. Furthermore, we try to make the fuzzing process more efficient by learning more about the target beforehand. The Large Language Model performs an analysis on the Control Flow Graphs of the software we want to fuzz. This analysis includes a ranking of functions which seem to be the most vulnerable. Using this analysis, another Large Language Model can write the fuzzing harness and custom mutator in a way we can specifically target these functions. Our experiments seem to show that this implementation is better than the current state of the art implementations in terms of discovering new bugs. Furthermore, the Control Flow Graph analysis seems to have a positive effect on the time efficiency. Surprisingly, the effect of the custom mutator seems to be minimal. However, more experiment runs are needed to confirm these results.

1 Introduction

According to the ABN Amro, one out of five Dutch companies were harmed by cyber attacks in 2024 [32]. These cyber attacks could do serious harm, ranging from financial losses to disruptions of public services. To keep our software safe, an effective method of finding bugs in applications is necessary.

There are multiple methods to find undiscovered bugs in software, often divided between *static application testing methods* (SAST) and *dynamic application testing methods* (DAST). With SAST, the source code or binary files are analyzed to find vulnerable code, whereby the application is not run. With DAST, the application is run, to see the effects of specific actions on the running application.

One DAST method is called *fuzzing*. As stated by the CAPEC, *"Fuzzing is a software security and functionality testing method that feeds randomly constructed input to the system and looks for an indication that a failure in response to that input has occurred."* [62]. On a fuzzing campaign, a fuzzer feeds random data to a target; data that the target does not expect. The data is fed to the target using a fuzzing harness, which can be seen as a 'bridge' between the fuzzer and the target. On the one 'side' of the bridge, the fuzzer obtains the random generated data from the mutator, responsible for mutating the data in unexpected ways. On the other 'side' of the bridge, the fuzzing harness calls the application, to see the effects of the generated input.

Using this process, the fuzzer tries to find other code paths than the 'happy path'. A 'happy path' is the path taken with a test case that uses known and expected inputs [13], and are test cases built to pass. These paths do often not cover edge cases. With fuzzing, inputs may be generated that do cover these edge cases, thus not following the 'happy path'. The goal of covering these edge cases is to find new 'buggy states' in the code. These 'buggy states' may result in a crash. The fuzzer is responsible for observing these crashes and saving the input that triggered the crash. All this found information can be used to find new bugs in the code.

Overall, fuzzing is a heavily researched topic in Cyber Security [62, 43, 48]. Several tools, such as Google's OSS-Fuzz, has shown the importance of fuzzing [35]. As shown on the documentation of OSS-Fuzz [35], the service has, as of May 2025, identified and fixed over 13.000 vulnerabilities and 50.000 bugs in over 1.000 projects. This indicates the importance of fuzzing in Software Security.

1.1 Current challenges of fuzzing

Fuzzing comes with some challenges, making it harder to implement a fuzzing solution. One of these challenges is that, at this moment, fuzzing requires a lot of manual work. For every different target in an application or library that should be tested, a different fuzzing harness should be created, which takes time and experience. Furthermore, for some targets, a custom mutator may also be necessary. This may be the case when a part of the application can only be accessed when the input has some specific structure. To ensure that the input is in this specific structure, a custom mutator can be written which enforces this. To write a successful custom mutator, one should have a understanding of the structure of the inputs, which also takes time and requires experience.

A second challenge in fuzzing is that the fuzzing campaign can take a long time and a lot of energy. As can be seen in the results in [27], it can take up to one week to find several crashes. Running a fuzzer for this long may not be feasible.

A third challenge in fuzzing is that it is hard to distinguish the results from a fuzzing campaign. The fuzzer returns all the inputs that caused a crash. However, from these inputs it is not clear what the severity of the crash is, and if it indeed results in a bug which should be fixed. Furthermore, it is hard to distinguish duplicates. It may be the case that two different inputs crash the same function, resulting in two observed crashes. However, this cannot be derived from the input only.

A fourth challenge is that there are not many benchmarks available. Currently, the most commonly used ones are LAVA-M [19] and Magma [27]. This makes it harder to evaluate a fuzzer and their performance in

identifying bugs.

These reasons combined results in that although the importance of fuzzing is shown, industry can be hesitant to implement a fuzzing implementation into their development operations.

1.2 Research Problem

Currently, there is a lack of automatic fuzzing tools. Fuzzing takes a lot of manual work and experience, resulting in fuzzing not being widely used in industry. The main goal of this thesis is to automate the fuzzing process, making it more approachable for industry to use fuzzing on their applications. Furthermore, using this research, we try to see if it is possible to fuzz more efficiently by learning more about the fuzzing target beforehand. To research this, we limit our scope to C [31] written applications and libraries. To address this, we set up the following Research Question: *How can we automate the fuzzing process of existing C-style programs, and generate custom mutators, fuzzing harnesses, and seed input files?*
With this Research Question, we have the following sub-research questions:

- Sub-question (SR1): 'How can we use Control Flow Graph and Large Language Models to identify vulnerable code?'
- Sub-question (SR2): 'What is the effect of the Control Flow Graph analysis on the efficiency of the generated custom mutator and harnesses?'
- Sub-question (SR3): 'How does the proposed implementation compare to current State of the Art implementations, in terms of efficiency?'
- Sub-question (SR4): 'How does Large Language Model generated fuzzing harnesses compare to human-written fuzzing harnesses, in terms of efficiency?'
- Sub-question (SR5): 'What is the cost breakdown between the analysis of the Control Flow Graphs and the generating of custom mutators, harnesses and seed files based on this analysis?'
- Sub-question (SR6): 'What is the effect of the custom mutator on the efficiency of the generated fuzzing harness with the Control Flow Graph analysis?'

With this work and the stated research questions, we try to address the first three issues, by making the whole fuzzing process an automated process. This will make fuzzing more approachable, since it minimizes the manual work and experience needed to start a fuzzing campaign. Furthermore, we use LLMs to analyze the generated CFGs of the source code of the software we want to fuzz to find vulnerable code, reducing the time and energy needed to randomly fuzz the starting point of the program. The chances that LLMs could have been used to analyze the CFGs in the same way, for the same goal, are slim. This makes it interesting to see how well the LLMs will perform with this task.

We focus on programs written in the C-language. Furthermore, we use the LLMs to generate the fuzzing harness and custom mutator, eliminating the manual work needed for this step. With these fuzzing harnesses and custom mutators, we try to target the vulnerable code directly, hopefully resulting in faster bug discoveries. We combined both the analyzing and generating in one tool, called LLM2Fuzz.

The structure of the thesis is as follows. First, we will discuss the required background needed to understand the terms in Section 2. All the related work will be discussed in Section 3. After that, we will discuss the architecture of the tool in Section 4. The experimental evaluation, with all the results, will be discussed in Section 5. Some future work recommendations will be given in Section 6. As last, the results will be discussed, the research questions will be answered and the conclusions will be drawn in Section 7.

1.3 Large Language Models for fuzzing

Large Language Models (LLMs) are currently used to perform several cyber security tasks. In [50], 29 different cyber security tasks were identified. Of these 29 different tasks, they found 119 papers covering 11

of these tasks where LLMs were used, such as bug detection and penetration testing. Furthermore, in [20], evaluations showed that LLMs can have a significant potential of transforming into cyber security frameworks. Another study showed that, while LLMs can have vulnerabilities such as 'jailbreaking', LLMs 'have a great potential for a wide range of cyber security tasks' [58]. LLMs can also be used in a defensive way, as shown in [51]. Here, LLMs are used to generate cyber security practice scenarios.

For fuzzing, Large Language Models are already used for fuzzing harness generation. Google's OSS-Fuzz-Gen is an extension built for Google's OSS-Fuzz [35], which leverages the use of LLMs to generate fuzzing harnesses for target applications. Furthermore, tools like LLamaFuzz [57] uses a LLM to fuzz structure-aware applications, by adding the LLM output to the input queue. Another way LLMs are used is to create a universal fuzzing application, targeting multiple application types in one. This approach is done using Fuzz4All [48].

These tasks shown that LLMs do have the potential to assist in cyber security related tasks. By showing this potential, we have confidence that we can also use Large Language Models in our research for the automation of the fuzzing process and improving the efficiency by analyzing the Control Flow Graphs.

1.4 Abbreviations

In this study, some abbreviations are used. All abbreviations can be found in Appendix B.

1.5 Acknowledgments

First of all, I would like to thank my supervisors from both Leiden University and TNO. Without your feedback, insights and support, this thesis would look different. Secondly, I would like to thank my girlfriend, my mother and my in-law parents. You were always there to offer your moral support and kept me going when things got rough. Without you, I am not sure how this thesis would have worked out; ik hou van jullie. At last, I would like to thank my friends, especially the ones I made last year. You were always there to listen, offer your support and you all brought the (sometimes much-needed) distractions. I am looking forward to many more 'mapo-drankjes'.

2 Background

In this section, we will discuss the required background knowledge needed to understand the rest of this work. We first will look at different type of fuzzers and how they differ from eachother. After that, we will take a look at the required components of a fuzzing campaign, which are the fuzzing harnesses and mutator strategies. As last, we will take a look at a small fuzzing campaign and what Control Flow Graphs are.

2.1 Type of fuzzers

In current fuzzing terminology, a difference is made between white, grey or black box fuzzers [2]. With a *white box fuzzer*, the fuzzer has access to all the information about the target we want to fuzz. This can include the source code, network diagrams and system architectures [18]. This type of fuzzing can be used to find vulnerabilities which are very hard to spot. With black box fuzzing, one does not have any knowledge of the inner workings of the application [2], making it harder to find deep buried vulnerabilities in the code. Grey box fuzzing is a combination of white and black box fuzzing [2]. One usually has some information about the target, such as the source code, but the network diagram or system architecture is missing.

In the case of white or grey-box fuzzing, we can use specific metrics to optimize our fuzzing process. Using these metrics, the fuzzer tries to obtain inputs which correspond to the highest score on a specific metric, such as code coverage [47] for binaries or endpoint coverage for APIs [7]. Furthermore, the fuzzer tries to find 'buggy states' in the application, often observed due to the application crashing. With these found crashes, cyber security researchers try to make the available software more secure.

This metric-driven fuzzing is harder with black-box fuzzing, due the fact that, as explained earlier, we do not know anything about the program apart of the binary. Therefore, it is harder to keep track of metrics, such as code coverage. Research for metric-driven fuzzing for binaries does exist, as explained in [44].

2.2 Fuzzing harness

To fuzz an application, the fuzzer needs an entry point to start, called the *fuzzing harness* [4]. The fuzzing harness can be seen as a wrapper over something you want to fuzz. For example, in C-terms, a fuzzing harness is the `int main()` of a fuzzing campaign and is responsible for converting the generated and mutated input to an input that will be accepted by the application we want to fuzz.

For example, we have function `foo`, which requires three parameters `int a`, `char b` and `double c`. Furthermore, our mutator returned 13 bytes of randomly generated data. In the fuzzing harness, we can then split this 13 bytes up in three junks of data: one of 4 bytes for the integer, one of 1 byte for the char and one of 8 bytes for the double. This way, we can call the function we want to fuzz with all the required information. The difference between the harness and the mutator is that the mutator is responsible for mutating the random data, while the harness can be seen as a 'bridge' between the fuzzer and the target application. The harness handles the random generated and mutated data and performs the right function call. An example of this fuzzing harness can be seen in Listing 1.

```
1  #include "example.c" // File containing unsafefuncion(int32_t a, char b, double c)
2  #include <stdint.h>
3  #include <string.h>
4  int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
5      // Check if we have enough size for one int32_t, one char and one double
6      if (Size < ((sizeof(int32_t)) + sizeof(char) + sizeof(double)))
7          return 1;
8
9      // Initialize the variables
10     int32_t a;
11     char b;
12     double c;
13     // Copy the size of an integer to a
14     memcpy(&a, Data, sizeof(int32_t));
```

```
15      // Skip the integer and then copy the size of a char to b
16      memcpy(&b, Data + sizeof(int32_t), sizeof(char));
17      // Skip the integer and char and copy the size of a double to c
18      memcpy(&c, Data+ sizeof(int32_t) + sizeof(char), sizeof(double))
19      // Call unsafe_function
20      unsafefunction(a, b, c);
21      return 0;
22  }
```

Listing 1: Example pseudo-code fuzzing harness.

Fuzzing harnesses can differ per application one wants to fuzz. A fuzzing harness for a binary will differ a lot from a fuzzing harness built to fuzz a REST endpoint. Furthermore, the fuzzing harness can be used to catch intended exceptions in the code. These exceptions are not found crashes and can result in false positives, and should not be reported by the fuzzer.

2.3 Mutators and structured input

Most mutators of fuzzers work by altering the data using a specific strategy. These strategies are needed to generate inputs that can reach new paths in the code. It may be the case that randomly generated data can explore a whole new code path. This code path can have several different sub-code paths, which requires a similar input, but have some small changes to it. Here is where the strategies come in: instead of waiting until we randomly generated a similar input, which can take a long time, we can mutate upon these already generated input files. This way, we ensure we keep the same structure of the input, but change small things to reach all the different paths inside this code path.

Multiple strategies exist, such as a mutation [56] or search-based [55] strategy. With a mutation strategy, the fuzzer starts with a corpus of seed files, and keep mutating these seed files into new inputs. These seed files are often mutated by algorithms like bit-flipping or splicing the input in different subsections. With bit flipping, random bits in the input bytes are flipped from 0 to 1, or 1 to 0. With splicing, we combine multiple generated inputs, splice them in half and combine them both with each other to create new inputs. With a search-based strategy, the fuzzer tries to execute a specific part of the code using various inputs [55]. It should be noted that this is not a guarantee that the specific part of the code will be executed using these inputs. With these types of problems, the fuzzer can use evolutionary algorithms [55]. With these evolutionary algorithm, we have a 'population' of input files, where each population differs a bit from the rest. The population which results in the most new code paths is seen as the 'strongest' population. The fuzzer will continue to create new generations based on this population, and is therefore called an natural evolution algorithm [55].

One limitation of the standard mutator strategies may be that an application can require some sort of specific, structured input. One example of this would be a PNG handler. As specified in the PNG specification [16], each PNG starts with the same eight bytes, namely `137 80 78 71 13 10 26 10`. With these eight bytes, a PNG handler knows that the input is a PNG file.

If we want to fuzz a PNG handler, we should make sure that every input we send to the handler, conforms to this specification. If not, the application will reject our input at the first check, preventing us from fuzzing the rest of the program. The chances of the fuzzer randomly guessing the right eight bytes, allowing is to fuzz further in the program, are extremely low. This way, we lose a lot of time and resources on fuzzing a small part of the program.

A fix for this problem is to use *custom mutators* [3]. With a custom mutator, we get more control on how we send the randomly generated input to the fuzzing harness. In the case of a PNG handler, we can hard code the first eight bytes to be the required specification, to make sure that we will at least pass the 'magic byte' check. We can then place the randomly generated data after these eight bytes, to make sure that we can fuzz the rest of the handler during our fuzzing campaign.

2.4 Seed input files

Several fuzzers rely on input seed files. These input seed files consists of valid (thus non-crashing) input for the application, which are 'fed' to the application using the fuzzing harness. These input seed files are used to start the built of a corpus [1], whereby a corpus is a set of collections using interesting input files. These interesting input files can be used by the mutator to find new, interesting paths in the code. An input seed file should cover as many code paths as possible.

2.5 AFLPlusPlus

AFLPlusPlus (or AFL++) [21] is a fuzzer which built upon AFL [25]. AFL++ is a gray-box fuzzer built to fuzz C and C++ type of programs. It supports multiple extensions compared to AFL, such as more mutation strategies and faster speeds. Furthermore, AFL++ supports libFuzzer [5] fuzzing harnesses.

2.6 Fuzzing campaign

To make the fuzzing process more clear, the whole process of a fuzzing campaign is explained here. The example is about a small C-program which will be fuzzed using AFL++.

Let's examine the following function in Listing 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
void unsafefunction(int32_t a, int32_t b) {
    if (a == 0) {
        if (b == 0) {
            printf("Hello world");
            return;
        }
        if (b == 1) {
            abort(); // crash!
        }

    }
}
```

Listing 2: Unsafe C-style pseudo program.

This small, minimalistic example shows that, when a is 0 and b is 0, 'Hello world' will be printed to the screen. When a is 0 and b is 1, the program will crash.

We can start by writing the fuzzing harness to address this function. The fuzzing harness can be found in Listing 3.

```c
#include "example.c" // File containing unsafefuncion(a, b)
#include <stdint.h>
#include <string.h>

// In here, Data is the mutated data, which we obtained from the mutator of the fuzzer
// Size is the size of the Data
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    // Check if we have enough size for 2 integers
    if (Size < (2*(sizeof(int32_t))))
        return 1;

    // Cast the first 4 bytes to a, second 4 bytes to b
    // We do not care about the data after 8 bytes
    int32_t a, b;
    a = 0;
    b = 0;
    memcpy(&a, Data, sizeof(int32_t));
    memcpy(&b, Data + sizeof(int32_t), sizeof(int32_t));
    unsafefunction(a, b);
```

```
    return 0;
}
```

Listing 3: Example pseudo-code fuzzing harness.

In this example, we first check if the size of the given data is large enough to obtain 2 integers. If this is not the case, we immediately return out of the harness. If we do have enough data, we `memcpy` the returned data to two integers and call the unsafe function.

After the creation of the fuzzing harness, we need to provide input seed files. A valid input seed file can be `00` or `11`. With `00`, we can print 'Hello world', and with `11`, nothing happens. We can provide these seed files to the fuzzer.

With the target, fuzzing harness and valid input seed files, we can start the actual fuzzing process. Since this is a very small, simple example, AFL++ can quickly find the crash. After the program has crashed, AFL++ will save the input that caused the crash (in this case, `01`) and will continue to look for other code paths.

When the fuzzing campaign is done or stopped, an output folder of all the inputs resulting in a crash will be returned. These inputs can then be used to further investigate the 'buggy states' and fix the found crashes and/or bugs.

2.7 Control Flow Graphs

Control Flow Graphs, CFGs, were proposed by Frances E. Allen, where *"A control flow graph is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths"* [10]. In this work, we focus on CFGs where the basic blocks contain assembly code. The CFGs reveals a lot of information about the code, such as the different code paths and functions used.

Nowadays, CFGs are mostly used for static analysis of source code, such as malware detection [14]. Using these CFGs, a programmer can obtain a deeper understanding of the mechanics of the code, especially of the whole control flow of the program. CFGs make a visual representation of all paths which can be taken in the program [10]. The same objective can be obtained with looking at the source code, but a CFG makes this more easily understandable.

In this research, we let LLMs analyze these CFGs. Since it can be hard for LLMs to analyze pictures, we use the DOT-language [26] to represent the CFGs in a text format.

As an example, examine the code in Listing 4.

```
#include <stdio.h>

int main() {
    int a = 0;
    if (a == 0) {
        printf("A is 0");
    } else {
        printf("A is not 0.");
    }
    return 0;
}
```

Listing 4: Example C program.

Using Clang, this can be converted to the CFG graph as shown in Figure 1.

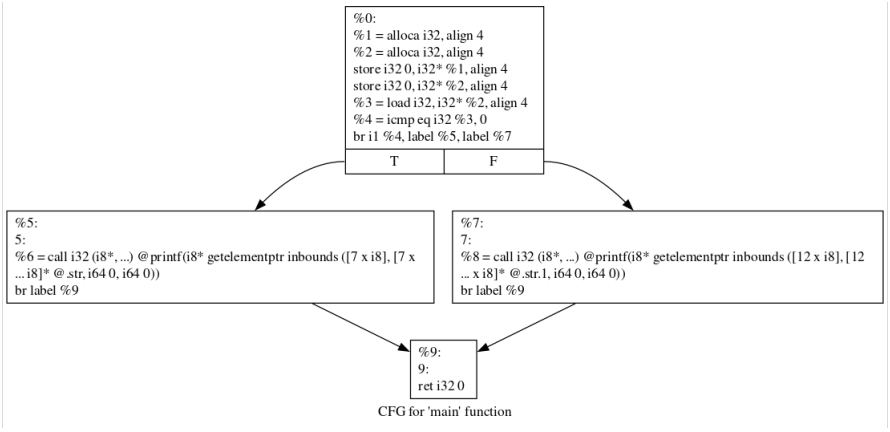

Figure 1: Small CFG example.

In Figure 1, we see the intermediate language where Clang converted the code to. Furthermore, we see the two branches: the `True` (if `a == 0`) and `False` (if `a != 0`) branch, showing us the whole Control Flow of the program.

3 Related Work

In this section, we will discuss related work in current state of the art fuzzers and fuzzer benchmarks. Furthermore, we will discuss some related work in bug triaging.

3.1 Fuzzing and type of fuzzers

At the time of writing, many different fuzzers exists. As mentioned in Section 2.1, fuzzers can be classified in white-, gray- and black-box fuzzers. Some white-box code fuzzers examples are SAGE [24], KLEE [17] and Driller[45]. SAGE is created by Microsoft and performs dynamic symbolic execution to speeds up the fuzzing process. KLEE is created at Stanford, which also performs dynamic symbolic execution and is built on top of the LLVM engine. Driller is created at UC Santa Barbara and is a guided, white-box fuzzer.

For grey-box fuzzing, some examples are AFL++ [21], libFuzzer [5] and T-Fuzz [43]. AFL++ is an enhanced version of Google's AFL [25] which includes multiple community patches. T-Fuzz works by transforming the fuzzing target to remove several checks, such as checks on magic bytes. This way, the fuzzer tries to improve the coverage of the fuzzer for the fuzzing target. libFuzzer is created by LLVM, and is a library for coverage-guided fuzzing.

While AFL++ is a grey-box fuzzer, it has the option to act as a black-box fuzzer with the QuickEmulator, QEMU mode [12, 21]. HonggFuzz can, just like AFL++, use the QEMU mode to fuzz as a black-box fuzzer.

As explained in Section 2.2, the fuzzing harness is an important component of the fuzzing process. Different fuzzers for different source programs are available. For example AFL++ cannot be used to fuzz REST APIs, while WuppieFuzz [7] can. It is also possible to fuzz other devices or systems, such as WhisperFuzz can fuzz processors [15] and the following paper [52] describes a way to fuzz the 5G protocol.

Apart of the mentioned fuzzers, a lot more exists. We refer the reader to survey papers [36, 54, 11] that analyze the literature in-depth.

3.1.1 Fuzzing with LLMs

As mentioned, in our work we focus on grey-box fuzzing for C code and we apply LLMs for generating the fuzzing harness, custom mutator and seed files. Existing research in the combination of fuzzers and LLMs are discussed here.

LLAMAFUZZ is a fuzzer, which tries to use the LLAMA Large Language Model to better fuzz grey-box targets by creating structure-aware inputs [57]. Their experiments show that, especially in structured-data fuzzing, LLAMAFUZZ performs better than other state-of-the-art fuzzers available. However, it should be noted that the LLAMAFUZZ fuzzer is evaluated on the MAGMA-benchmark [27], and that it may be the case that the MAGMA data is used in the training set of the LLAMA model. The authors did also test LLAMAFUZZ on real-world programs. These experiments showed that the LLAMAFUZZ fuzzer outperforms AFL++, reaching 27.19% more code coverage on average in absolute numbers than AFL++. In 10 out of the 15 targets evaluated, LLamaFuzz obtained a significant higher code coverage than AFL++. Another tool which tries the same, but with OpenAI's ChatGTP [41], is ChatFuzz [29]. With ChatFuzz, an average improvement of 11.04% is shown upon the standard AFL++ implementation in the terms of code coverage. It should be noted that both LLAMAFUZZ and ChatFuzz use different targets to benchmark their fuzzer.

'Fuzz4All' [48] is a tool that tries to create a language independent fuzzing tool. Usually, fuzzers are built for a specific programming language, such as Python or C++. However, by using LLMs, the authors tried to create a tool which can fuzz for multiple languages without explicitly building support for it. The tool can create language specific fuzzing harnesses, which can both address the starting point of the application or a specific function. Experiments showed that, in a 24 hour fuzzing campaign, Fuzz4All achieved the highest coverage compared to other state-of-the-art fuzzers [48]. These are different fuzzers than the fuzzers

explained above. All the used fuzzers can be found in [48]. Google's OSS Fuzz Gen is an extension built for Google OSS Fuzz. With OSS Fuzz [35], Google tries to distribute the fuzzing of open-source software. This makes large-scale fuzzing more approachable and as a result, the software more secure.

To address the manual creation of the fuzzing harnessess, Google created OSS-Fuzz-Gen to automatically create fuzzing harnesses for the target applications. Using the Fuzz Introspect [42] tool, an LLM could investigate which function is not addressed enough. An LLM is then used to generate a fuzzing harness to address this specific function, trying to improve the code coverage of the fuzzer.

The biggest difference between Google OSS-Fuzz-Gen and the work presented in this thesis is that OSS-Fuzz-Gen generates harnesses using the line coverage metric for optimization. By using this approach, it only aims to maximize the achieved line coverage, but can miss important code paths. In our work, we aim at synthesizing harnesses based on a more fine-grained approach. Using this approach, we try to reach specific code paths of interest in a target function.

3.2 Fuzzer benchmarks

There are several benchmarks available to evaluate the performance of fuzzers, mainly targeted to fuzz C and C++ programs. One of them is Magma, which is a 'ground-truth fuzzing benchmark' [27]. Magma works by altering a set of vulnerable code, placing so-called oracles to check if a certain bug is reached or actually triggered by the fuzzer. The difference between the two is that with 'reached', the context of the bug is reached, and with 'triggered', the fault actually occurred. When the canary is reached or triggered, the function *MAGMA_LOG* is called, with some information on which bug is reached or triggered. This function keeps track of the total number of reached and triggered bugs. Magma supports 138 different bugs in 7 different projects and 22 different fuzzers out of the box. By keeping track of all the reached and triggered bugs, Magma can make a ranking out of the fuzzers [27].

The bugs used in Magma are real bugs which were once found in the target libraries, but patched in future versions. To let these bugs work with a newer version of the library, the bugs are 'patched up' and injected into a newer version of the library. This means that some of these bugs can be quite old, with the oldest bug implemented in Magma is found in 2011 [27].

Another benchmark for fuzzers is UniFuzz, which is, unlike Magma, metrics-driven [34]. Some metrics used by UniFuzz are the quality of bugs and the overhead the fuzzer uses. The quality of the bug is determined by matching a bug to a Common Vulnerabilities and Exposures (CVE) number and use the Common Vulnerability Scoring System (CVSS) score and by using the output of the Exploitable tool in GDB [22].

Besides Magma and Unifuzz, ShadowBug [60] can be used as a benchmark for fuzzers. This tool generates certain types of bugs, such as buffer overflows, and injects this into an existing code base.

In this work, we benchmarked LLM2Fuzz against Magma. We chose Magma for various reasons. The first one is that the injected bugs are real bugs injected in real life programs. This way, we have a benchmark which is very close to the real world. Secondly, Magma also keeps track of when we reached the vulnerable code path, but not triggered a bug. While this is not necessarily useful when fuzzing in real life, it does show information about the capabilities of the LLM to identify vulnerable code paths, which is relevant to answer one of the sub research questions.

3.3 CFG, fuzzing and vulnerability detection

In current research, Control Flow Graphs are used for both vulnerability detection and fuzzing. In detection, CFGs are mainly used for malware detection. One study [14] uses CFG analysis for malware detection on IoT devices. Another study uses CFGs [39] to classify malware based on weighted CFGs. We refer the reader to a survey paper [37] about CFG analysis for malware detection to read an in-depth research on this topic. For software vulnerability detection, there is a study [59] about identifying software vulnerabilities by using

an modified version of CFGs

Currently, CFGs are used in some research with fuzzing. One study states that generated CFGs by reverse engineering tools can help by creating fuzz targets for binaries [53]. Furthermore, another study [61] shows the effect of CFGs used for guided fuzzing. Another studies [49] shows the effectiveness of CFGs for the seed scheduler.

These studies show that CFGs can be an successful addition to both vulnerability detection and fuzzing. Our approach differ from the current research due the fact that we use the Control Flow Graph to identify vulnerable functions. The Control Flow Graphs are analyzed by Large Language Models, which summarize the results and make a ranking of the most vulnerable functions. This analysis is then used to instruct the fuzzing harness and custom mutator to specifically target these functions. Using this approach, we aim to make the process more efficient. To the best of our knowledge, this approach is not yet implemented by other research.

4 Architecture

In this section, we will discuss the design and implementation of the custom mutators and the feedback loop of the fuzzer, which are the main contributions of this thesis.

4.1 Design

The application is designed in a modular way, which results in an easy to maintain and extendable project. Each stage of the program is created in their own class. Furthermore, each stage has its own LLM client, which has its own instruction prompt for that specific task. This way, each LLM can fully focus on one specific task. All the LLMs are instructed to a specific output format, such that the LLMs and the code can interpret each other.

In Figure 2, a diagram is shown of the designed tool. All different paths are explained in this section.

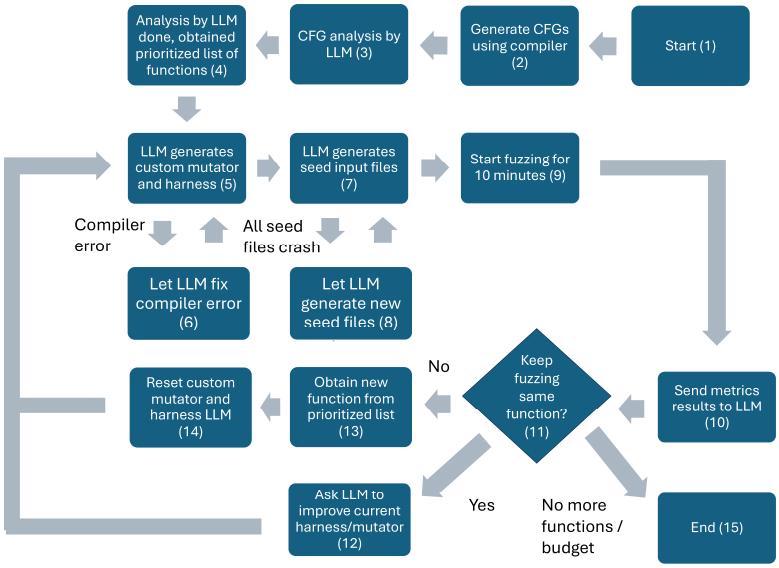

Figure 2: Overview diagram of the design.

4.1.1 Control Flow Graphs: Phase One

In the first phase, we generate and analyze the CFGs of the source code. This is shown in the diagram with the first four blocks. The CFGs of the source code we want to fuzz are generated using the compiler, and sent to the LLM. The CFGs are compiled on the same machine we start the fuzzing campaign later. The LLM is asked to perform an analysis on the CFGs and return some data about the CFGs back to the user. With this analysis the application obtains a deeper insight in the code. It is, using the CFGs, possible to obtain all the different code paths the code can take, which can be used to be more efficient in targeting a specific function or code path.

The context window of LLMs is currently limited, which makes it infeasible to store the whole CFG of the program in the LLMs memory. Therefore, the CFGs are iteratively created and summarized by the LLM. The LLM is prompted to keep the most important and vulnerable functions in the summary, as can be seen in the prompts in Appendix A.1. After each summary, a new set of CFGs is sent to the LLM, along with the summary.
By following this approach, a lot of valuable information can get lost. However, due to the restriction of the limited context window, this is at the time of writing the only usable method. We try to minimize the amount of tokens needed for the CFGs. These methods will be explained in Section 4.2.2.

The following data is returned from the LLM. This data is returned for each function.

- Overall explanation - Small explanation of what happens in the CFG in general. This is the summary.
- Name of each function we want to fuzz.
- Parameters per function, where the different data types and some small details about what the parameter does are returned.
- Vulnerability rating per function - Rating from 0-100, where 0 is probably not vulnerable and 100 is probably fully vulnerable.
- Explanation per function - Small explanation on why the function has this vulnerability score.
- Paths to function - All the different code paths the program can take to get to this particular function.

The different paths to the function and the vulnerability rating are especially interesting. With all the possible paths to a function listed, the application can try to find all possible combinations where it is possible to manipulate input data which can lead to a crash. With the vulnerability rating, the application can order the functions based on the vulnerability score. These can be functions that, for example, use other unsafe functions on raw pointers.

The LLM is prompted to respond in JSON, where the format is described in Listing 5. By adhering to this JSON format, the application can easily parse the data and use the results for the next stages.

```
{
  "explanation": "explanation in string form here",
  "functions": [
    {
      "name": "example_function_1",
      "parameters": [
        {
          "type": "i32",
          "detail: "Example detail"
        }
      ],
      "vulnerabilityRating": 30,
      "explanation": "why is the function vulnerable",
      "pathsToFunction": [
        "main",
        "function2"
      ] // Functions we can take to get to that function.
        // Useful if we want to fuzz a private function,
        // and therefore cannot call the function directly
        // in our harness.
    }
    ]
}
```

Listing 5: JSON format of the response of the LLM on the CFG analysis

As last, the CFG class is also responsible of keeping track of which function we try to fuzz. If the LLM decides that we have fuzzed the function enough, we continue to the next function. The LLM decides this based on the number of found crashes, code coverage by lines and path coverage by comparing the taken paths with the CFG.

4.1.2 Custom Mutator and Harness: Phase Two

In Phase Two, the custom mutator and harness are generated by the LLMs. In Figure 2, this phase is represented in block 5 and 6.

It would be possible to let the LLM generate input files and use these in a queue, which can then be sent to the fuzzer. However, an efficient binary fuzzer executes at least 1000 executions per core per second [30]. If we were to use this approach, we would need to mix the input files of the LLM with an already existing strategy, such as bit flipping. This would make it harder to examine the effect of the LLM generated inputs on the target, since the distinction between input files generated by the LLM and an existing strategy would be harder to notice. However, as discussed in Section 3.1, there is some research which implemented this design. The prompts for this instruction can be found in Appendix A.2.

In stead of letting the LLMs generate the mutated input files, we decided to let the LLM generate a custom mutator. A custom mutator is responsible for the mutation strategy and is often used in programs which require specific input, such as a PNG handler [9]. We re-use this strategy to craft inputs which, in as little time as possible, can efficiently reach all the different code paths. The same LLM agent creates the custom harness, to ensure that they are built to work with each other. By using a custom mutator, we still have control over the inputs given to the program, but we maintain the same execution speeds we normally have while fuzzing.

The following data is returned from the LLM to the code. This information is only from this LLM agent, and not from any other data sources in the project.

- Harness - The code for the custom harness.
- Custom mutator - The code for the custom mutator.
- Harness extension - File extension of the harness source code file.
- Mutator extension - File extension of the mutator source code file.
- Explanation - Small explanation of the harness and custom mutator which are generated.

The data is returned in JSON in the format shown in Listing 6. We included the explanation to the JSON output to make the output of the LLM more explainable to the responsible end-user.

```
{
  "harness": "<harness code>",
  "custom_mutator": "<mutator code>",
  "harness_extension": "<harness extension>",
  "mutator_extension": "<mutator extension>",
  "explanation": "<small explanation>"
}
```

Listing 6: JSON format of the response of the LLM for the harness and custom mutator

The custom mutator and harness are built in a feedback loop, which will be explained further in Section 4.1.4.

4.1.3 Seed generation: Phase Three

The third phase of LLM2Fuzz is the seed generation, where we ask the LLM to generate several seed files. The seed files rely on the source code of the fuzzing harness, which is why this is the third phase. In Figure 2, this phase is shown in block number 7 and 8.

AFL++ is a mutation based fuzzer [8], which means that it tries to obtain the highest code coverage

using mutations of previous inputs. To start, the fuzzer needs a couple of valid seed files, which at least covers some of the code. These seed files should *not* crash and should be as small as possible.

The LLM is asked to generate input files based on static analysis of the code and the CFGs. This way, the LLM can generate seed files that cover most of the code paths inside the function we want to fuzz. This analysis is a part of the static analysis of the project. The LLM is asked to generate as many different seed files as deemed necessary. At least one of the seed files should not crash, as this is an requirement from AFL++. If the LLM returned less than one valid seed file, the LLM is asked to fix the invalid files, such that they do not crash anymore. The prompts for this instruction can be found in Appendix A.3.

4.1.4 Fuzzing process: Phase Four

Phase 4 of LLM2Fuzz is the fuzzing itself. This phase is represented in block 9 till 14 in Figure 2.

The fuzzer is the last part of the application, where the program tries to actually find crashes. The fuzzer is initialized with the custom mutator and harness and run with the inputs generated in the previous steps.

We did not create our own fuzzer. Instead, we built a 'wrapper' over an existing fuzzer, AFL++. This cover monitors the results of the fuzzer and keeps track of the custom mutator, the harness and input seed files required for AFL++.

Concurrently with the fuzzer, a monitoring program is run, which keeps monitoring the progress of the fuzzer. The monitor keeps track of the number of found crashes and the code coverage. The code coverage is obtained by using the line coverage as a proxy metric. This way, we can determine if there are lines in the code that are not taken and use this information to improve the harness and mutators. We ask the LLM to compare the code coverage to see if any lines inside the function we want to fuzz is not yet, or not enough, taken, to ensure targeted fuzzing. Based on this information, the LLM can decide to alter the custom mutator to take specific paths.

These monitoring details are sent to the LLM in a customizable interval. The LLM can then decide one of following the three options:

1. Keep fuzzing the same function with the current settings; everything looks promising, so continue.
2. Keep fuzzing the same function with other settings; thus, change the custom mutator and/or harness.
3. Start fuzzing another function; the fuzzer focused on this function enough and moved it's interest to another function.

This 'feedback loop' tries to ensure that the fuzzer will fuzz the program as efficient as possible. The LLM can determine if certain 'edge cases' or different paths in the code are not taken, and rewrite the custom mutator and harness in such a way that these cases or paths will be taken. The prompts for this instruction can be found in Appendix A.2.

It may be the case that the LLM gets confused by a trivial error and a human intervention could be preferable. Therefore, the application has the possibility for the human to 'step in' and speak to the LLM itself. This form of 'human in the loop' ensures that the application does not spend too much resources on trivial problems. The option for the human to step in is presented after 5 failed tries in compiling the custom mutator or harness. In our final experiments, we disabled this option, to test the fully automatic capabilities of the LLM to fuzz the target programs.

4.2 Implementation

The implementation of the code is done in Python 3.12.3. We decided to use Python since this makes the communication with different LLMs fairly easy, especially due to the OpenAI API. It is possible to use different models from OpenRouter [6] using this API, making it straight forward to implement different

models. Furthermore, Python is a convenient language to interconnect multiple components with each other, which is also used in this implementation.

4.2.1 Use of Large Language Models

The LLMs (GPT 4.1, Google Gemini 2.5 Pro Preview and Meta LLama 4 Maverick) are accessed using an abstract base class. In the base class, the functions `_add_message_to_prompt` and `reset_model` are implemented. The first function adds a message to the prompt history which will be sent to the LLM. The second function resets the prompt history, but ensures that the system prompt stays. This function can be useful if the application decides to switch to creating a custom mutator for another function; the LLM does not need the history of the other functions. By removing the previous prompts, the application reduces the input tokens, which leads in a more efficient use of the LLM.

Furthermore, the class has one abstract method, called `talk_to_llm()`. This function should be implemented on each different LLM ecosystem, since the way of communicating with the LLM differs per provider.

One thing to note about the way we used the LLMs in LLM2Fuzz is that we do not use the `structured outputs` attributes, which are for example available in OpenAI LLMs [40], but also an option in Ollama models. This option ensures that the output will be in a specific format, such as JSON. We noticed during our own experiments with OpenAI's models that, with the structured outputs enabled, the quality of the harnesses and custom mutators decreased significantly. Therefore, we decided to put our expected output format in the prompt, as explained in Section 4.1.

Using these abstract base classes without structured outputs, we can easily implement new LLMs in the code, making the code modular.

4.2.2 Control Flow Graphs: Phase One

The CFGs of the code are generated using the `clang` [33] and `opt` [33] tool. The `clang` tool generates LLVM intermediate representation [33] language, which can be used as a input for the `opt` tool. We compile the code to the LLVM Intermediate Representation language using the `-O3` flag, which is used for optimizations such as loop unrolling and vectorizing. These optimizations results in less code, thus smaller CFGs. Furthermore, we strip the CFGs of all enters and unnecessary whitespaces. This way, we minimize our CFGs, resulting in fewer tokens.

The CFGs are generated in the DOT-language [26]. Using this language, we have a textual representation of a CFG, which can be sent to the LLM.

4.2.3 Custom Mutator and Harness: Phase Two

The custom mutator and harness are generated by the LLM. It may be the case that the LLM is not capable of generating a correct custom mutator and harness in the first try. If this is the case, the compiler error is printed to the user and sent to the LLM, with the intention to fix the mistake. There is an option to, after a customizable number of tries, prompt the user to talk to the LLM itself to fix trivial errors. Using this option, the user can keep the 'human-in-the-loop'.

The prompts to this LLM client also includes the code and the output of the `ls` command in the code directory. We noticed that, when the LLM has access to the code, the LLM is less likely to generate incompatible code, for example by forgetting to include certain header files. Furthermore, the output of the `ls` command is used to know the filenames of the headers the LLM should include. All the analysis is still done on only the CFGs, and the Custom Mutator and Harness client is not responsible for deciding which function or path we want to fuzz.

4.2.4 Seed generation: Phase Three

The seed files for the fuzzing campaign are generated by the LLM. As explained in Section 4.1.3, the seed files should not crash. Our program checks the seed files by executing the single seed file and checking the return status code of the fuzzer. If more than a customizable number of seed files returns in a crash, we send these files back to the LLM and ask the LLM to fix these inputs.

4.2.5 Fuzzing process: Phase Four

The program is implemented as a multi-threaded solution to run both the fuzzer, fuzzer cover and monitoring system at the same time. This way, the program can monitor while the program keeps fuzzing. The communication between the threads is done using queues, since queues in Python are thread safe [23].

The `AFL_CUSTOM_MUTATOR_ONLY` environment flag is enabled to ensure that the fuzzer only uses the custom mutator it generated, and not the built in fuzzing strategies.

The program uses afl-cov [38] to keep track of the coverage. This information is, in combination with the CFGs, periodically sent to the LLM to decide if the custom mutator and harness are good enough. If this is not the case, the LLM can generate a new custom mutator or harness and prompt the program to restart the fuzzing campaign with the new custom mutator or harness.

By using this approach, the program combines static and dynamic analysis to ensure the most efficient way of fuzzing, where the program will only focus on relevant crashes. This is checked by the LLM by comparing the code coverage obtained from afl-cov after each iteration. The static analysis is done based on the CFGs and the dynamic analysis is done based on the results from running the created custom mutators/harnesses and input files.

5 Experimental Evaluation

In this section, we will discuss the Experimental Evaluation of the fuzzing application. We used an existing benchmark, Magma [27], to show the effectiveness of both tools.

With these experiments, we want to measure the effectiveness of LLM2Fuzz in detecting vulnerable code and writing specific harnesses and custom mutators. To measure the effectiveness, Magma monitors how many times the fuzzer reach and trigger specific bugs in a fuzzing campaign. Furthermore, we want to see which models perform the best with a fixed price limit of $10. This price limit is chosen based on the budget obtained from the client, which was overall 100$. To make a fair division between the experiments and ensure that every model got a chance to run, we gave each experiment a budget of 10$. Our results show the time it took to trigger or reach the bug, both with and without the time it took for the program to start fuzzing.

5.1 Set up

Magma works by containerizing every combination of a target program and a fuzzing engine. The *fuzzing campaign* runs inside the container. In the container, the target code is fetched from an remote repository. The bugs for the targets are *patched* in the code, and after that the code will be compiled into a library, which can be used to fuzz. For every target, Magma includes a fuzzing harness, which can be used in the benchmark. Furthermore, Magma contains a tool set to create and maintain these fuzzing campaigns, and keep track of the reached and triggered bugs. All the injected bugs are numbered, with for example `PNG001` the first bug they injected into the LibPNG base. All the injected bugs with their respective numbers can be found on the Magma site [28].

To let LLM2Fuzz work with the Magma benchmark, we had to alter the way Magma usually containerize the fuzzing campaign. LLM2Fuzz works by analyzing the CFGs of the source code. We want to test the ability of the LLMs to recognize vulnerable code and write specific mutators and harnesses for this. However, with the Magma canaries written in the code, it may be the case that we give away the bugs to the LLM by detecting these canaries. To avoid this, we include the source code twice: once with both the canaries and the bugs and once with only the bugs. The first one is used to compile to a library, while the second one is used for the CFG analysis. The logic in both codes is exactly the same, but the function call to the canary monitoring mechanism is missing. This way, the LLM cannot link the Magma canary to an injected bug.

All our experiments are run on a server with the following configuration:

- Ubuntu 22.04.5 LTS.
- 12 core 2.4GHz CPU.
- 8GB of RAM.
- 100GB of SSD storage.

5.2 Results

In this subsection, we will discuss the results of the experiments with the fuzzing implementation.

5.2.1 Libraries and LLM models

For our experiments, we ran our code on two targets of Magma; LibPNG and LibTIFF. We decided to use these two projects since both of these targets are designed to be used as a library within C or C++ projects. Due to resource limitations, we were not able to run the benchmark for all Magma targets. Magma polls every oracle every 5 seconds, which means that we get the updated reach and trigger count every 5 seconds.

In these experiments, three different LLMs are benchmarked: OpenAI GPT-4.1, Google Gemini 2.5 Pro

Preview 2.5 and Meta LLama Maverick 4. The first two models are, at the time of writing, state-of-the-art models, where the last one is not. As a result, the first two models are a lot more expensive. The prices can be seen in Table 1.

Model	$/1 million input tokens	$/1 million output tokens
GPT 4.1	2	8
Google Gemini 2.5 Pro Preview	1.25	10
Meta Llama 4 Maverick	0.15	0.60

Table 1: Table showing the costs per million input and output tokens.

All three models have a context window larger than 1.000.000 tokens, which is necessary for these experiments.

Not all experiments ran for the same time, due to various reasons. All the different run times and reasons can be seen in Table 2.

Target/model	Time	Reason
LibPNG - GPT 4.1	1h 3 min	Ran out of money
LibPNG - LLama 4 M	1h 51 min	Crashed (LLM did not follow instructions)
LibPNG - Google Gemini 2.5 Pro	13 min	Ran out of money
LibTIFF - GPT 4.1	48 min	Ran out of money
LibTIFF - LLama 4 M	4h 33 min	Ran out of money
LibTIFF - Google Gemini 2.5 Pro	24 min	Ran out of money

Table 2: Table showing all the runtimes of the experiments.

5.2.2 Graphs

Apart of the raw data in the tables, the data is also presented in a graph. The graphs will be explained here.

Figures 3, 5, 7 9 and 11 show the cumulative number of bug events until a certain timestamp. A bug event means that we either reached or triggered a bug. Each dot means that there is a difference in one of the triggered bugs compared to the previous timestamp. We only plotted a dot, if there is a change in the cumulative number of bug events on that timestamp. Furthermore, the dotted lines show all the different milestones in the graph. All the milestones are laid out in Table 3.

Color	Explanation
Blue	We generated and analyzed all the CFGs.
Orange	We start fuzzing a function for 10 minutes.
Green	We start fuzzing a new iteration of the function for 10 minutes.
Red	We switch our focus to a new function.

Table 3: Explanation of the different milestones in the graphs.

We also included histograms of all the bug events, namely Figures 4, 6, 8, 10 and 12. In these histograms, we see the frequency of a specific bug event between two different timestamps. For example, in Figure 4, we see that we have a bin of around 10^6 at 500 seconds. This means that between the 495 and 500 seconds, PNG006 is reached around 10^6 times.

Reaching a bug is not an useful metric in real-life fuzzing. If we do not know anything about the bug, reaching the bug does not tell anything about existing; only triggering can cause a crash, which we can

monitor. However, we still decided to show these results. Reaching a bug, in this context, can mean that the implementation did target a function which is vulnerable. This may show that the CFG analysis is effective in recognizing vulnerable code, but that it is harder to let the LLMs write the harness or custom mutator to actually trigger the bug.

5.2.3 LibPNG

For LibPNG, both the GPT-4.1 and Meta LLama Maverick 4 have reached several bugs, whereby LLama Maverick 4 also triggered one bug several times. Our budget of 10$ per run was not enough for the Google Gemini 2.5 Pro Preview 2.5 to start the actual fuzzing process, so we do not have any results for this LLM. The overall results can be seen in Table 4.

Model/bugs	Bugs reached	Bugs triggered	Which bugs
GPT 4.1	1	0	`PNG006`
Llama 4 Maverick	2	1	`PNG002` (R+T), `PNG006` (R)
Google Gemini 2.5 Pro	0	0	None

Table 4: Number of bugs reached and triggered in LibPNG.

GPT-4.1 In Figure 3, we can see the results of the run on LibPNG with GPT-4.1. In the graph, we see that the implementation first reached `PNG006` at around 450 seconds. This delay can be explained by that the CFG analysis, creating of the mutator and harness and the creating and testing of the seed files takes time.

Furthermore, we observed some interesting behavior. We see that around the 2000 second timestamp, we move from the `PNG006` bug to the `PNG005` bug. The logs show that, as intended, we switch at this point to another identified vulnerable function, which explains why the `PNG005` bug is reached after this point, while the `PNG006` bug is not. However, as explained in Section 4.1.3, we need at least one non-crashing seed input to start the actual fuzzing process. The logs shows that the LLM was not capable of doing this for the `PNG005` bug; all the times the bug is reached, is due input files that caused a crash. However, we see in the graph that the bug is only reached, and not triggered. Nonetheless, the LLM did switch to another vulnerable function and this shows that the LLM identified at least two vulnerable functions in the code.

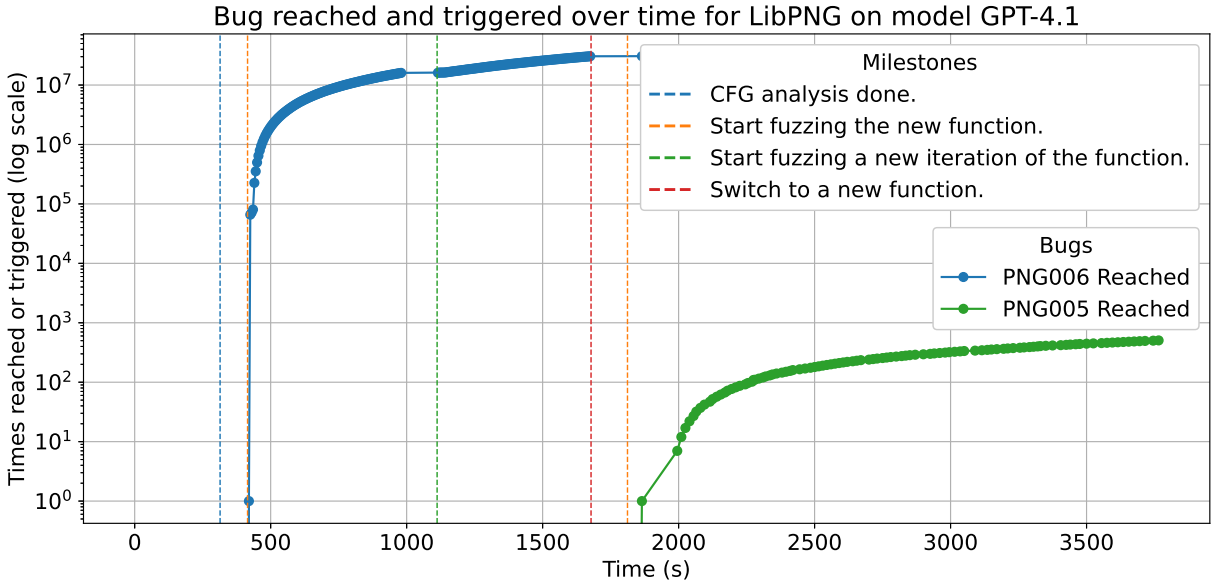

Figure 3: Bug reach events on LibPNG with GPT-4.1. No bugs were triggered.

In Figure 4, we can see a histogram of when we reached a bug. The reached bugs in the graph are a delta of the previous timestamp. We see several breaks between the found bugs. These breaks can be explained by the time it takes to talk to the LLM, fixing compiler errors and creating new seeds files.

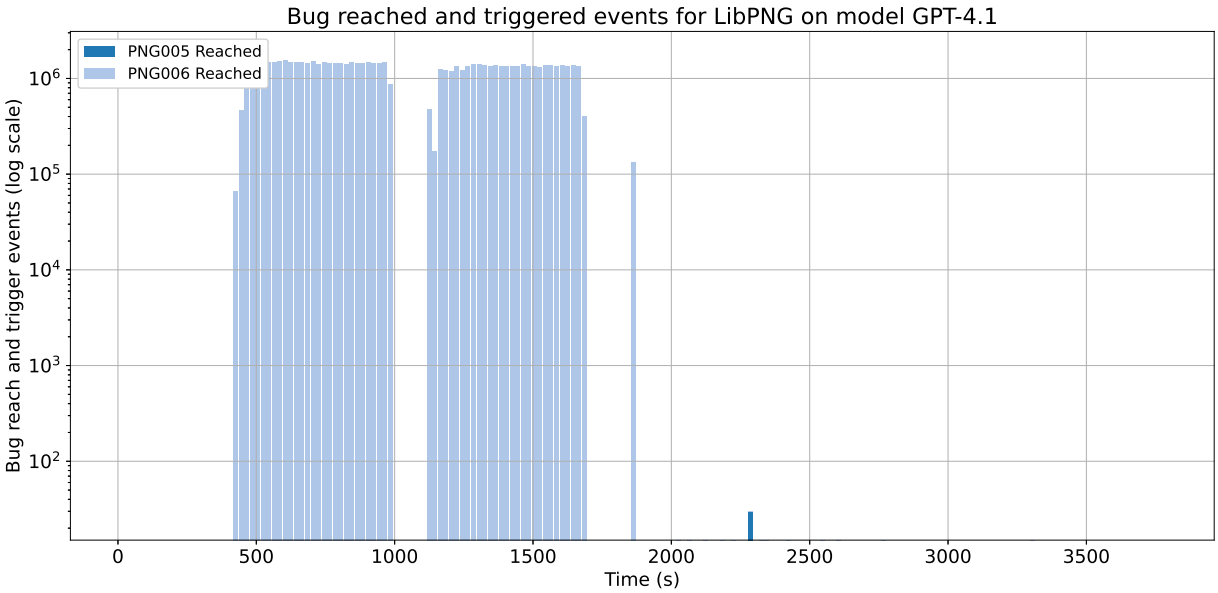

Figure 4: Histogram for bug events on LibPNG with GPT-4.1

Meta LLama Maverick 4 In Figure 5, we see the total number of bugs reached and triggered on LibPNG with the Meta LLama 4 Maverick model. Firstly, it should be noted that the triggered `PNG002` and reached `PNG006` are the exact same line. To make them visible on the graph, we added a small buffer to the triggered line. This means that, every time the `PNG002` bug was triggered, the `PNG006` bug was reached as well. One explanation for this may be that after reaching the `PNG006` bug, the input triggered the `PNG002` bug. This may be the case if the vulnerable code is called after we went through the vulnerable code of `PNG006`, but our input did not satisfy the requirements to actually trigger the `PNG006` bug.

Furthermore, we see that, starting from around 800 seconds, the model can reach and trigger various bugs. The graph looks like that the number that `PNG002` is reached is a multitude from the number that `PNG006` is reached, which supports the fact that `PNG002` and `PNG006` are connected with each other.

As last, we see that we sometimes reach a bug before we start fuzzing. This observation is explained by the same behavior we see in Section 5.2.3; some seed files, created by the LLM can reach or trigger a bug, which will be picked up by the benchmarking tool.

It should be noted that, due to the LLM not following the right instructions, LLM2Fuzz crashed after around 3 dollar used. Nonetheless, the LLM was capable of reaching two bugs and triggering one bug.

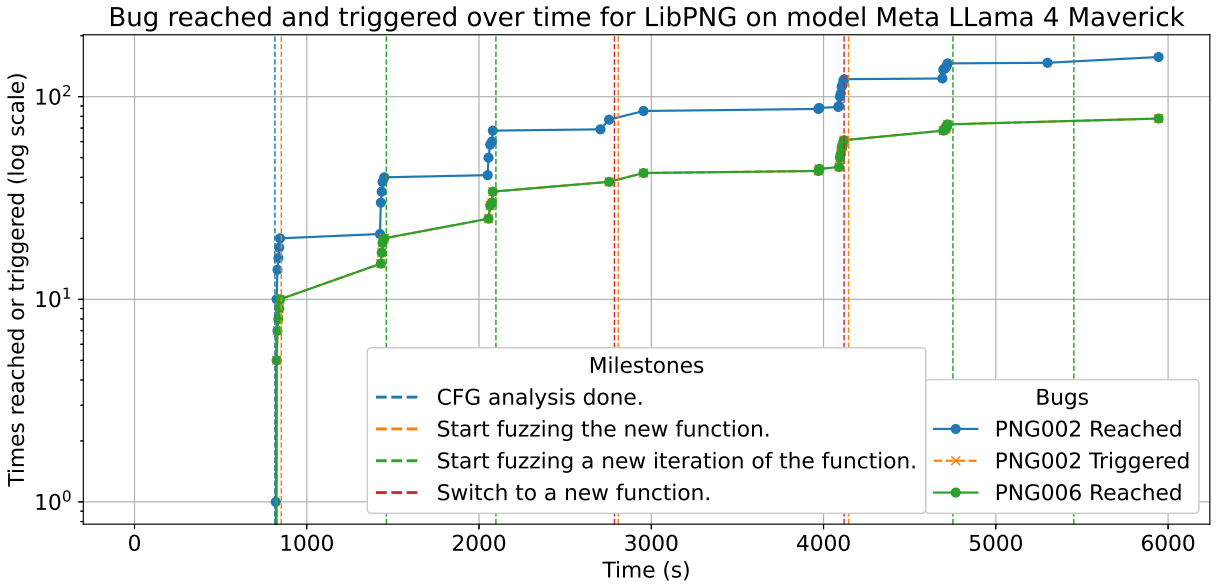

Figure 5: Bug reach and trigger events on LibPNG with Meta LLama 4 Maverick

In Figure 6, we see the histogram of the experiment with LibPNG and the Meta LLama 4 Maverick model. In here, we also see a lot of breaks between the encountered bugs, with the same explanation as the previous histogram; talking to the LLM, recompiling and fixing the seed files takes time, which is time we cannot use to fuzz.

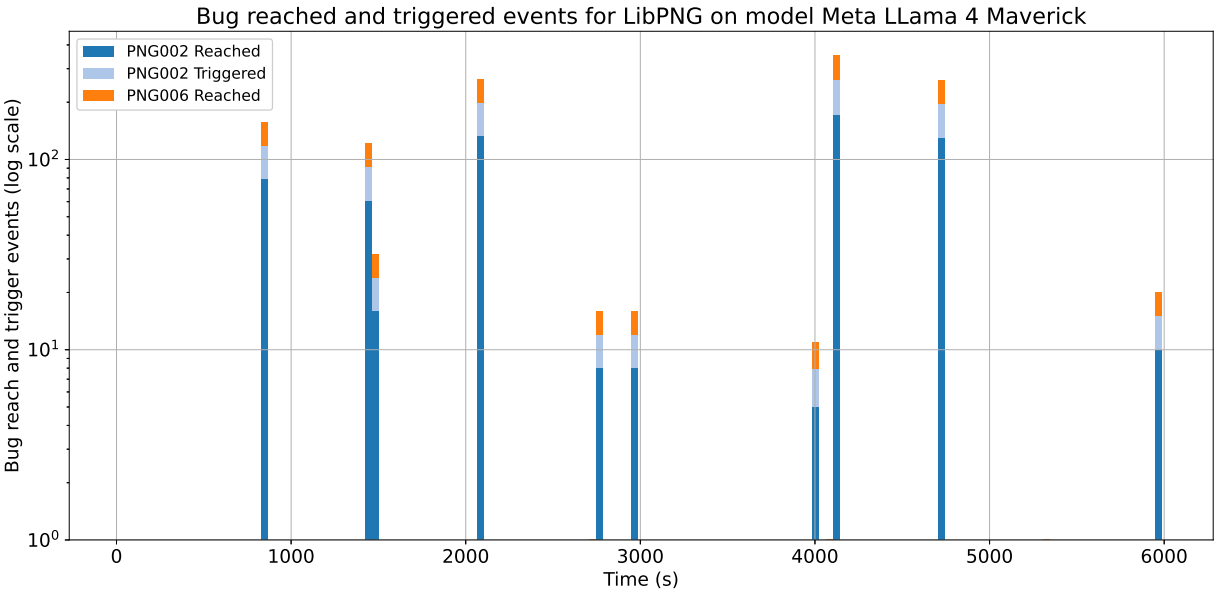

Figure 6: Histogram for bug reach and trigger events on LibPNG with Meta LLama 4 Maverick

Google Gemini 2.5 Pro Unfortunately, the 10$ for the experiment was not enough to actually start fuzzing with the Google Gemini 2.5 Pro model, resulting in no results for this experiment.

5.2.4 LibTIFF

For LibTIFF, our approach did only reach one bug with the Meta LLama Maverick 4 model. For GPT-4.1, our approach did not reach or trigger any bugs in the target. Our budget of 10$ was also not enough for the Google Gemini 2.5 Pro Preview to start the actual fuzzing campaign. The overall results can be seen in Table 5.

Model/bugs	Bugs reached	Bugs triggered	Which bugs
GPT 4.1	0	0	None
Llama 4 Maverick	1	0	`TIF012`
Google Gemini 2.5 Pro	0	0	None

Table 5: Number of bugs reached and triggered in LibTIFF.

GPT-4.1 OpenAI's GPT-4.1 did not reach or trigger any bugs for the LibTIFF target. After looking at the coverage data, we saw that the CFG analysis focused on a non-vulnerable function at first. Furthermore, the coverage of the targeted function is quite low; the custom mutator did not succeed in reaching all the different code paths in the target function. After trying to improve the harness and mutator, the coverage became even lower. In a long run, there would be a chance that the LLM can fix these errors or switch to another function. However, due to budget restrictions, the implementation did not reach this point.

Meta LLama Maverick 4 In Figure 7, we see that we managed to reach one bug at the end of the campaign. According to the logs, we can see that this is around the same time that we switched to another function. Unfortunately, the implementation did not get the chance to improve the harness and custom mutator, due to the fact that the budget was all used up almost immediately after the find of this bug.

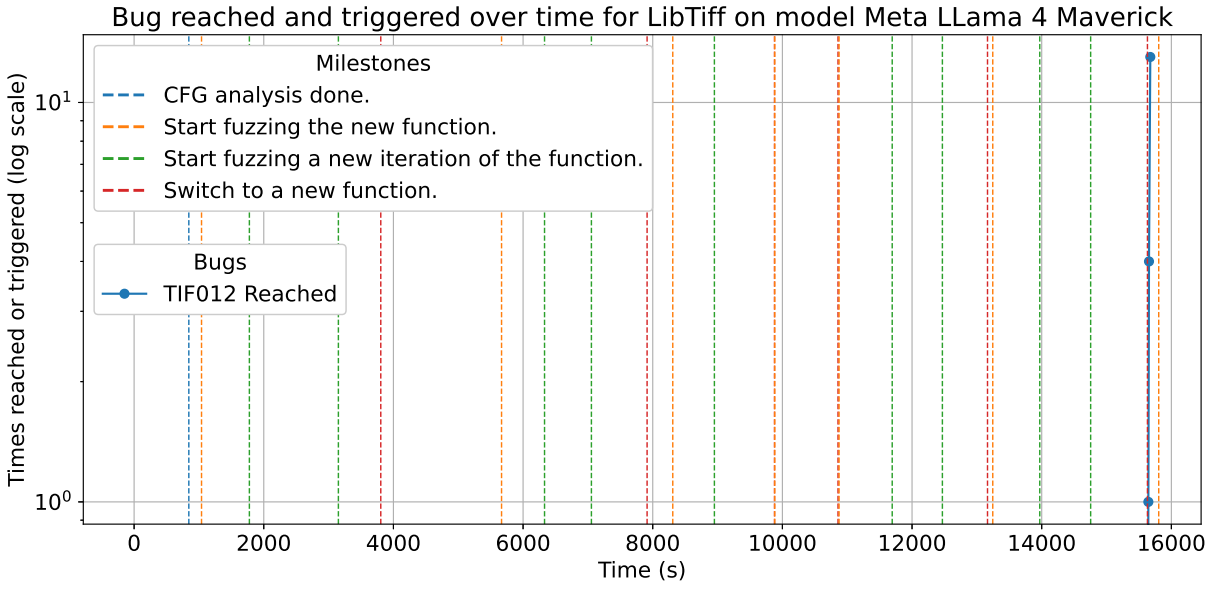

Figure 7: Bug reach and bug trigger events on LibTIFF with Meta LLama 4 Maverick

In Figure 8, we see the histogram of the experiment with LibTIFF and Meta LLama 4 Maverick. As expected, there is only one peak, since we only reached a bug at the end of the campaign.

Target/costs (% total cost)	Costs CFG	Costs fuzzing
LibPNG - GPT 4.1	\$2.57 (25.7%)	\$7.43 (74.3%)
LibPNG - LLama 4 Maverick	\$0.31 (3.1%)	\$9.69 (96.9%)
LibPNG - Google Gemini 2.5 Pro	\$6.63 (66.3%)	\$3.37 (33.7%, ran out at seed generation)
LibTIFF - GPT 4.1	\$6.77 (67.7%)	\$3.23 (32.3%)
LibTIFF - LLama 4 Maverick	\$0.34 (0.34%)	\$9.66 (96.6%)
LibTIFF - Google Gemini 2.5 Pro	\$10 (100%)	\$0 (0%)

Table 6: Table showing the costs breakdown of the CFG vs the rest of the program

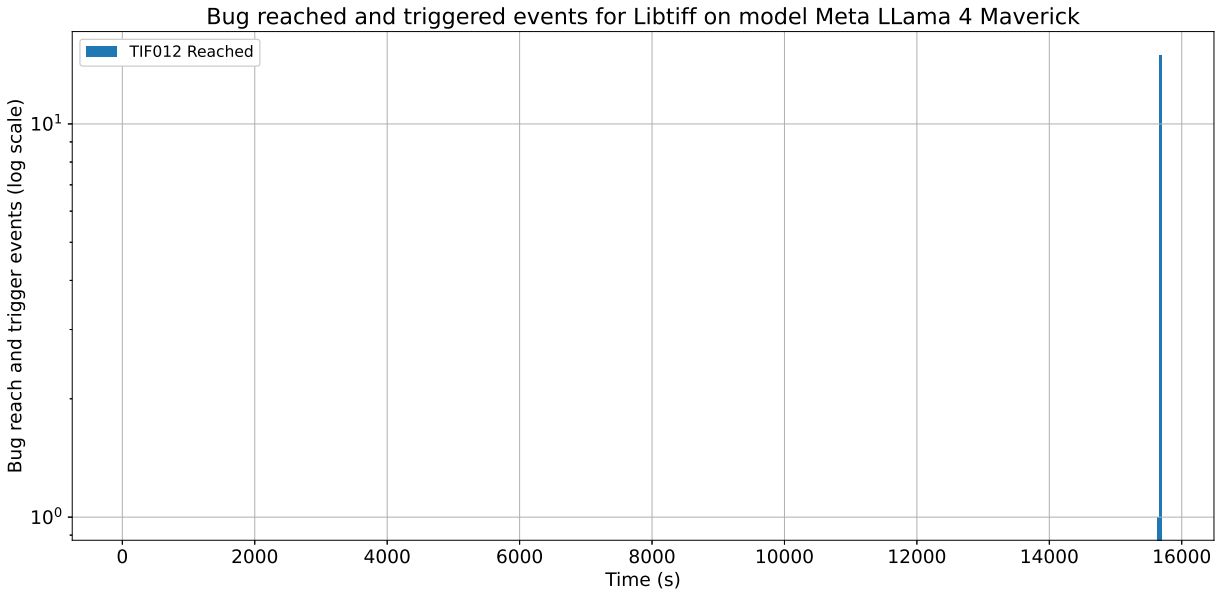

Figure 8: Histogram for bug reach and bug trigger events on LibTIFF with Meta LLama 4 Maverick

Google Gemini 2.5 Pro Unfortunately, the 10\$ for the experiment was not enough to actually start fuzzing with the Google Gemini 2.5 Pro model, which means that there are no results.

5.2.5 Cost breakdown

Since the number of distinct bugs is quite low, it is hard to make a claim about the cost effectiveness of the tool. Therefore, we give an overview of the costs breakdown between the CFG analysis and the actual fuzzing.

As can be seen in Table 6, there is a big difference between several models. Google's Gemini 2.5 Pro is the most expensive one, while LLama 4 Maverick is the cheapest option.

5.2.6 Comparison to current fuzzers

To compare our approach with the current state-of-the-art implementations, we compared our results with AFL++ and OSS-Fuzz-Gen. Since LLM2Fuzz is built upon AFL++, we compared our results to fuzzing campaigns done with AFL++, to see if our approach improved the results.

For our first experiment, the comparison with AFL++, we re-used the fuzzing harnesses with are included by the Magma benchmark. These harnesses are also used by Google's OSS-Fuzz service and are human written. By comparing LLM2Fuzz against these harnesses, we can see the difference between human written

and LLM generated harnesses. Since there are no human written custom mutators available in Magma, the included strategies in AFL++ are used. We compared this fuzzing campaign against the results shown in Section 5.2.3 and 5.2.4.

For our second experiment, the comparison against OSS-Fuzz-Gen, we compared our approach with the current state-of-the-art in LLM based harness generation. With this experiment, we generated custom harnesses with OSS-Fuzz-Gen which are used in the fuzzing campaign. The goal of this comparison is to see the effect of the Control Flow Graph analysis on the creation of fuzzing harnesses and custom mutators by LLMs. Since OSS-Fuzz-Gen can only generate fuzzing harnesses, and not custom mutators, the included strategies in AFL++ are used. We compared this fuzzing campaign against the results shown in Section 5.2.3 and 5.2.4.

We let AFL++ run for the same time of the longest run of our other experiments ran before it ran out of money. In the first experiment, we used the human created harnesses, while in the second experiment, we used the OSS-Fuzz-Gen generated harnesses. This means that we ran AFL++ for 1 hour and 39 minutes for LibPNG and 4 hours and 21 minutes for LibTIFF.

AFL++ Our experiment for AFL++ on both LibPNG and LibTIFF showed that the fuzzer did not reach or trigger any bugs for both targets in their respective time frame.

The writers of Mamga also performed experiments with their benchmark on a set of fuzzers. The results can be found in [28]. These results show that, for both LibPNG and LibTiff, more bugs are reached and triggered than with LLM2Fuzz.

For LibPNG, we see that in most fuzzers, most bugs are reached quickly, at around 10 seconds. It can take a longer time to actually trigger the bug.

For LibTIFF, we see that LLM2Fuzz performed noticeably worse than the fuzzers tested with the experiments from Magma. The existing fuzzers can trigger and reach many bugs in a short time span, while LLM2Fuzz is not capable of doing that.

It should be noted that for most of these fuzzers, the fuzzing campaign ran for a longer period of time than our fuzzing campaign. Furthermore, it should be noted that these fuzzers are instructed to get a as high as possible code coverage, as soon as possible. LLM2Fuzz is instructed to focus on one function or vulnerability at a time.

AFL++ and OSS-Fuzz-Gen To test the implementation against a current state-of-the-art approach, the results of the first experiments are compared to OSS-Fuzz-Gen. As explained in Section 3.1.1, OSS-Fuzz-Gen is a tool used to generate fuzzing harnesses. In this experiment, OSS-Fuzz-Gen generated multiple fuzzing harnesses for the target programs. The results can be seen below.

For LibPNG, Google's OSS-Fuzz-Gen generated three different harnesses. All these three harnesses are tested. The harnesses were created to work out-of-the-box with Google's OSS-Fuzz and are altered to work with the Magma benchmark. The results can be seen in Table 7.

Harness	Bugs reached	Bugs triggered	Which bugs
oss_png_readpng	1	0	`PNG006`
oss_fuzz_png_image_finish_read	0	0	None
png_handle_iccp	1	0	`PNG006`

Table 7: Number of bugs reached and triggered by OSS-Fuzz-Gen generated harnesses.

The only bug reached by two of the three harnesses is `PNG006`. No bug is triggered.

For LibTIFF, OSS-Fuzz-Gen tried to generate different harnesses, but only two of them compiled. Both these harnesses are tested, and just as the harnesses for LibPNG, altered to work with the Magma benchmark. The results can be seen in Table 8.

Harness	Bugs reached	Bugs triggered	Which bugs
tiffopen	0	0	None
tiffopenext	0	0	None

Table 8: Number of bugs reached and triggered by OSS-Fuzz-Gen generated harnesses.

Both with tiffopen and tiffopenext, no bugs are reached or triggered.

5.2.7 Ablation study

In this research, multiple different design choices could have an impact on the performance. These design choices are as follows:

- The summarization of the CFGs - The quality of the summarization can have an impact on the ability to create the fuzzing harnesses and custom mutators.
- Giving the LLMs the ability to decide when to stop fuzzing a particular function - The ability of the decision making of the LLMs can have an input on the fuzzing process.
- Do not use the 'human in the loop' function - We assume that the LLMs are capable of fixing every compiler error on it's own.
- Use the number of crashes and code coverage as metrics for the LLM decision making - It may be the case that other metrics can work better than these two.
- Fuzz for 10 minutes per iteration - It may be the case that, even with the aim to improve the efficiency, 10 minutes is still not long enough.
- Focus on functions based on the CFG analysis - It may be the case that analyzing the source code works better than analyzing the CFG analysis.
- Using the LLM to generate the custom mutator, instead of creating the input files itself - It may be the case that LLMs are not (yet) capable of generating good functioning custom mutators.

Since all these choices can have an impact on the performance, we performed ablation experiments to measure the impact of two of these choices. It was impossible to ablate all the different design choices. Therefore, we performed two, namely the use of the CFG analysis and the use of custom mutators. We decided to experiment with these two, since these two design choices are particularly new in the research. For further research, it is interesting to study the impact of the other design choices in this design.
To identify the impact of the different components in the application, we performed two experiments where we left something out. In the first experiment, we did not use any CFG analysis, and let the process run purely based on the included source code. With this experiment, we can see the effect of the CFG analysis on the fuzzing results.
In the second experiment, we reused the same CFG analysis used in the original experiment in Section 5.2.4, but did only create a custom harness and not a custom mutator. With this experiment, we can see the effect of the custom mutator on the process. Furthermore, we can compare this result to the results of OSS-Fuzz-Gen, to see the impact of the CFG analysis on harness generation. We re-used the CFG analysis to make sure that we can detect the difference between the use of a custom mutator or not, and make sure that the CFG analysis is not better or worse than in the previous experiment.

For the ablation study, we decided to build further upon the best performing model, which is Meta LLama Maverick 4 on the LibPNG target. This combination is the only combination which managed to trigger a bug. As explained earlier, while reaching a bug is interesting for benchmarks, in real-life, only triggering

a bug would result in useful results. Therefore, we decided to continue with the experiment setup which actually triggered the bugs.

In both the ablation experiments, we see the same behavior as found in Section 5.2.3: sometimes, a bug is reached or triggered right before we start fuzzing a new iteration or function. This may be due the fact that, before the implementation starts fuzzing, the implementation generates new input seed files. These seed files can reach or trigger a bug, which is picked up by Magma, and reported back in the graph.

Experiment 1: No CFG analysis To perform the ablation without the CFG analysis, the code base had to be altered. With this approach, we only sent the source code files to the LLM, and let the LLM itself decide which function it would want to fuzz. For this decision, no CFG analysis is used.

All the CFG references in the prompts are deleted and the code is altered to not use any generated CFGs.

The results can be seen in Figure 9. As can be seen in the figure, we do trigger and reach some bugs, starting from around the 2.000 seconds time mark. However, in comparison to Figure 7, the experiment where we did use the CFG analysis, we see that it does take longer to reach the first reached and triggered bug. Furthermore, we see that we trigger and reach it less than when we did use the CFG analysis, but even with these lower numbers, the number of triggered bugs is still sufficient for further analysis. The conclusion is that the main improvement of the CFG analysis, is the timing.

It should be noted that, just as the previous experiment, the implementation crashed after around 4.000 seconds due the LLM not following the right instructions.

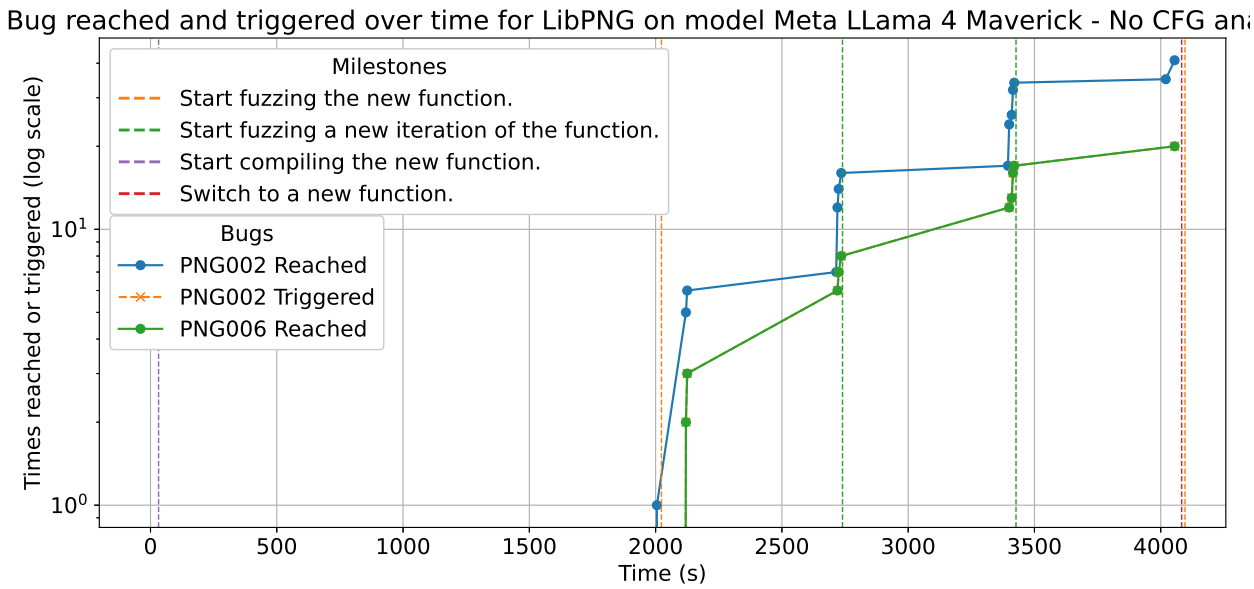

Figure 9: Bug reach and bug trigger events on LibPNG with Meta LLama 4 Maverick without a CFG analysis

In Figure 10, we can see the histogram of the ablation study without the CFG analysis. Using this graph, we can more clearly see that the number of reached and target bugs is very low, and that it takes a lot longer without the CFG analysis to reach or target the bugs than it takes with the CFG analysis.

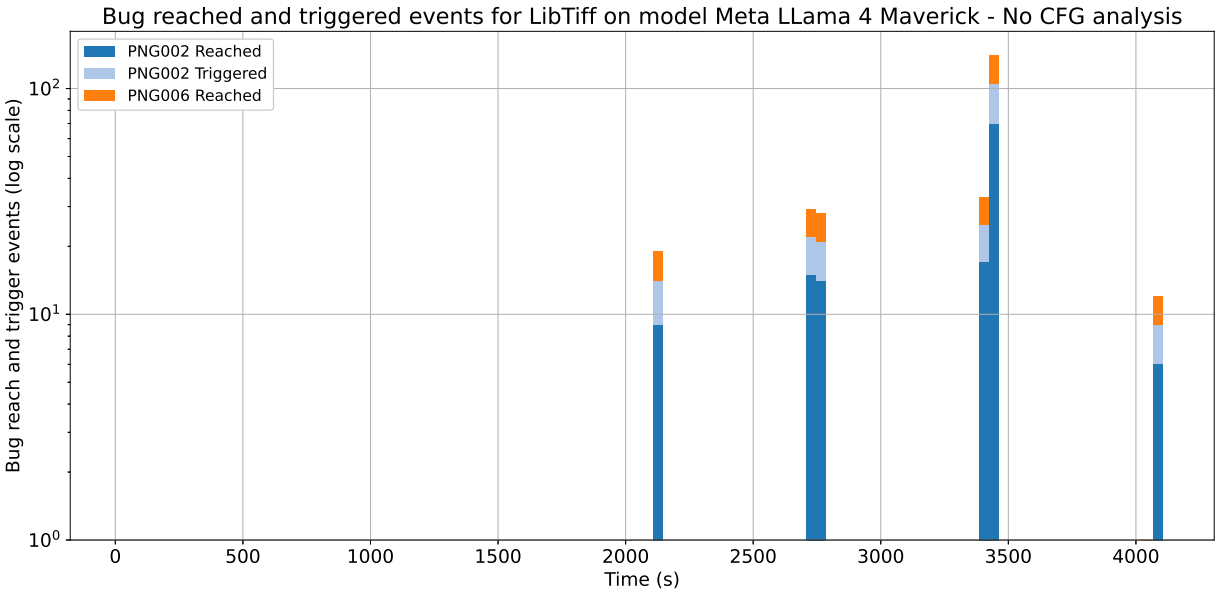

Figure 10: Bug reach and bug trigger events on LibTIFF with Meta LLama 4 Maverick - No CFG analysis

Experiment 2: No custom mutator To perform the experiment without the custom mutator, the code also had to be altered. We deleted all the references to a custom mutator in the code, and altered the LLM prompts to only generate a fuzzing harness and not a custom mutator.

As explained at the start of this Section, we re-used the CFG analysis we obtained in the first experiment. This explains the fact that we can start fuzzing quite quickly at the start of our fuzzing campaign. However, as seen in Figure 11, we see that we can reach and trigger bugs quicker without the custom mutator. Furthermore, we can see that we can reach a certain bug, `PNG006`, far more without a custom mutator than with a custom mutator. This experiment shows that the effect of a custom mutator to target specific input paths, with this target, is minimal.
It should be noted that, just as the previous experiment, the implementation crashed after around 4.500 seconds.

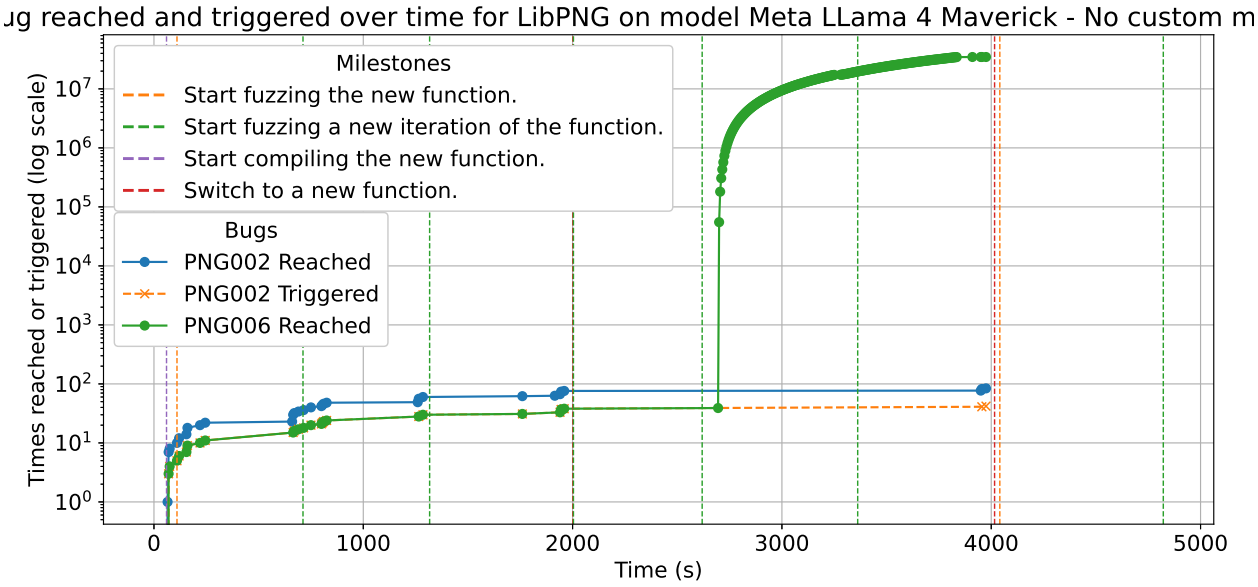

Figure 11: Bug reach and bug trigger events on LibPNG with Meta LLama 4 Maverick without a custom mutator

In Figure 12, we can see the histogram of the experiment without the custom mutator. In here, we can see that we can reach and target the bugs earlier than with a custom mutator, with a big spike around 3.000 seconds. In the logs, we see that this is about the time when we switched to another function, explaining the big spike in the reached bugs.

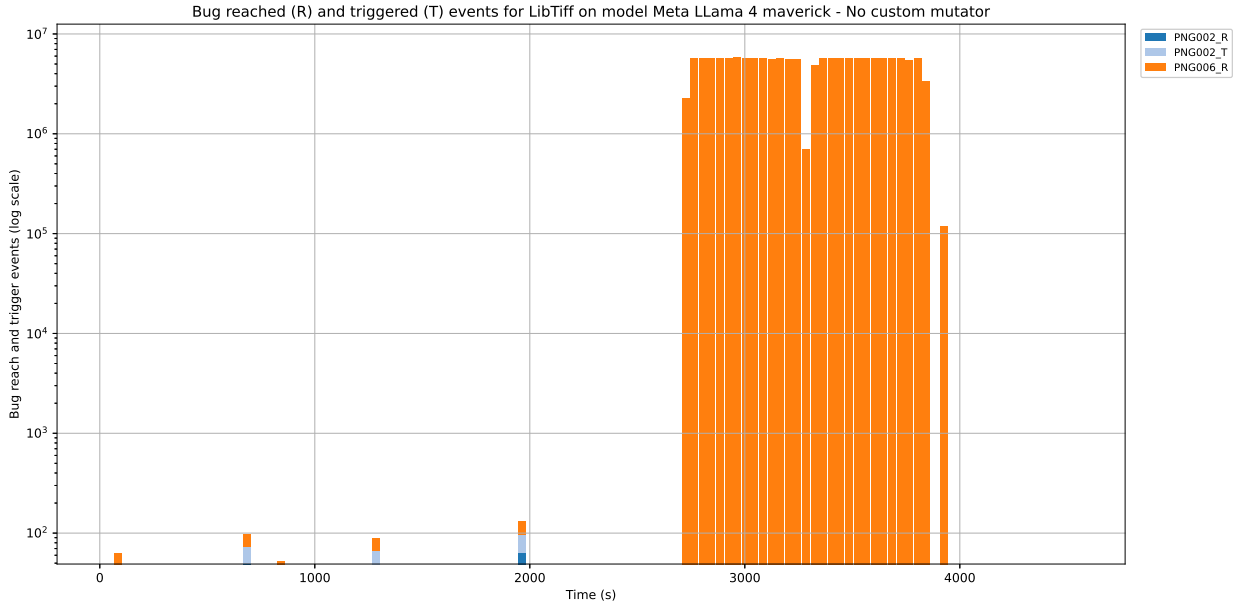

Figure 12: Bug reach and bug trigger events on LibTIFF with Meta LLama 4 Maverick - No custom mutator

6 Limitations and Future Work

In this section, we will discuss the limitations of this research. Furthermore, we will provide several different Future Work recommendations, which can be used to further explore fuzzing in combination with LLMs.

The main limitation of our work is that, due to budget limitations, we were unable to run our experiments more than once. This reduces the reliability of our research, and it is an interesting further research topic to extend the experiments to multiple runs. With these multiple runs, we can gain more confidence in the findings. Furthermore, it would be interesting to see how the non-deterministic nature of LLMs would work with these type of tasks, in terms of reproducibility.

In addition, our results demonstrated an unexpected result. Without the custom mutator, the implementation managed to gain more bug events for reaching the bugs than without the custom mutator. This is unexpected, especially since we thought that a file format such as PNG would benefit from a custom mutator. We would like to explore this in the future.

At the start of the project, we also tried to automate the bug triaging part of a fuzzing campaign. We aimed at creating a large enough dataset using our fuzzer, which we could then use to validate our triage tool. Due to the limited results of our fuzzer, we were not able to create a large enough dataset for this experiment. Therefore, it is an interesting open question to experiment with the triage tool in the future, to see if it is also possible to automate this part.

Furthermore, as explained in Section 4.1, another limitation is the limited context window of the LLMs. Currently, we have to summarize the CFG analysis due to the limited context windows of the current LLMs. We think it would be very interesting to see how LLM2Fuzz behaves when we have access to LLMs that can fit the whole CFG structure in its memory. We think that a lot of valuable information gets lost in this summarizations, which could result in a worse result.

An extra limitation is that we do not dynamically test our generated harness and custom mutator. Currently, the implementation assumes that the harness always works, and just tries to fix the input seed files if we encounter any crashes. This is not a valid assumption and can waste resources in debugging this. It would be interesting to research the same approach as Google's OSS-Fuzz-Gen, where the fuzzing harness is dynamically tested before the fuzzing campaign is started.

In our current experiments, we used one type of LLM for all the different tasks. LLM2Fuzz has the option to use two different LLMs for the CFG analysis and the creation of the custom mutators and harnesses. It would be interesting to perform an experiment were we would use transfer learning of the LLMs. This could be achieved by letting a state of the art model analyze the CFGs and let a faster or cheaper non state of the art model generate the custom mutators and harnesses.

While LLM2Fuzz is capable of detecting and triggering some vulnerable code, it may benefit from a semi-automatic approach. In this semi-automatic approach, the LLM with the CFG analysis can make the prioritized list of vulnerable functions and a start of the fuzzing harness and custom mutator. After this, a human can perform optimizations based on the CFG report. This implementation may still be more efficient than fuzzing with the current state of the art implementations.

We also think it is interesting to see if we can support reversed binaries using this LLM2Fuzz. Using several Reverse Engineering tools, such as Radare2 [46], it is possible to generate Control Flow Graphs of binary applications. Using this, it would be possible to research the option to also fuzz binaries.

As last, we are interested to see how LLM2Fuzz would perform on obfuscated code. It may be an interesting experiment to check if and to which extent LLM2Fuzz can still identify vulnerabilities, when the vulnerable code is obfuscated. Especially reverse engineered binaries can be quite heavily obfuscated, making it an interesting future research experiment in combination with the future research experiment to also

support binaries.

7 Discussion and Conclusion

In this section, we will discuss our results we obtained from our experiments. Furthermore, we will answer the Research Questions as stated in Section 1 and give the conclusions.

With our experiments, we have seen that LLM2Fuzz is capable of detecting vulnerable code and can write custom mutators and harnesses to fuzz these vulnerable functions. However, as the Results show in Section 5.2, we see that there are some differences between models and targets.

First, we see that more bugs are reached and triggered for LibPNG than for LibTIFF. For LibPNG, `PNG002` is reached and triggered, while `PNG005` and `PNG006` are reached. For LibTIFF, only TIF012 is reached in one run. One explanation for this may be that LibPNG is more widely used than LibTIFF. Due to the wide use of LibPNG, it is possible that the LLMs have more information of the inner workings of these libraries, and could therefore have a deeper understanding of the workings of the library. Another explanation may be that while PNG is a heavily restricted format, TIFF has many optional fields. These optional fields results in a lot of different code paths that can be taken, and making it harder to identify and target a specific vulnerable code path. Nonetheless, the fact that the implementation can target these functions and reach these bugs, shows that it seems to be possible to use Control Flow Graph and Large Language Models to identify vulnerable code. Therefore, we can answer SR1 by that we can use Large Language Models to analyze the Control Flow Graphs, and ask the Large Language Models to return a list of the most vulnerable functions based on this analysis. Using this approach, we can use Control Flow Graphs and Large Language Models to identify vulnerable code.

Our results show that LLM2Fuzz, which let LLMs generate fuzzing harnesses and custom mutators based on the analysis of the CFGs by another LLM, works better than human written harnesses on an AFL++ campaign. While LLM2Fuzz reached three different bugs and managed to trigger one single bug, a run with the human created harness did not trigger or reach any bug at all. While this difference is small, the ablation study shows that the use of the CFG analysis seems to indicate that we can reach and trigger the same bug in the half the time.
The creators of Magma also performed experiments with various fuzzers and the Magma benchmark. In these experiments, the existing fuzzers performed better than the experiments we did with their harnesses, and better than the experiments with LLM2Fuzz. This difference can be explained by the servers used. The configuration of their servers for the fuzzing campaign is better than the configuration of the servers we used for our experiments, which meant that their fuzzing speed was faster. For our conclusion, we use the results of the experiments from our server, to make the comparison as fair as possible. With this information, we can answer sub-question SR2 with the answer that the Control Flow Graph analysis seems to have a positive effect on especially the time it takes to reach a vulnerable code path.

Furthermore, our results show that LLM2Fuzz works better than the OSS-Fuzz-Gen generated harnesses. The OSS-Fuzz-Gen generated harnesses only managed to reach the `PNG006` bug. No other bug is reached or triggered. With this information, we can answer sub-question SR3 with the answer that LLM2Fuzz seems to perform a bit better than current state of the art implementations.

LLM2Fuzz managed to trigger one bug. When fuzzing real life programs, triggering a bug is the only important thing; reaching a bug does not give any information about the bug, since it does not results in a crash. This results in that we can answer SR4 as that the implementation seems to have a positive effect on the efficiency, compared to human-written fuzzing harnesses, but that there is room for improvement on triggering the bugs.

One explanation for this performance may be that, due to the summarizing of the CFGs, a lot of useful information is lost. The context windows of the LLMs are currently not big enough to store the whole CFG in their context. A second explanation may be that the LLMs are (currently) not good enough in analyzing the CFGs to identify harder to spot vulnerable code paths. This may improve when the LLMs will get bigger and 'smarter'. A third explanation may be that the fuzzing campaign was too short to find

other bugs. We instruct the LLMs to generate a fuzzing harness and custom mutator to target the function which the LLM thinks is the most vulnerable. Due to our budget, it was in most cases not possible to fuzz the program for a long time, resulting in mostly fuzzing the same function in the fuzzing campaign. It may be the case that, when having more budget, the implementation can more often switch to other functions and find more bugs in there. As last, we observed that some fuzzing campaigns got stuck on a non-working harness. The harness is not dynamically tested; the implementation assumes that when the harness compiles, the harness is correct and will work. Furthermore, LLM2Fuzz requires to have at least one working seed input file to start fuzzing. If we do not have any working seed input file, the LLM is asked to recreate the input seed files, until we satisfy this requirement. This can result in an infinite loop, which can be fixed by first dynamically testing our harness before going to the next phase.

One could argue that, since the injected bugs which are implemented by Magma can be quite old, the LLMs have encountered these bugs in their learning phase and creates the priority list of the CFG analysis based on their memory. While it is true that most of the bugs already existed before the cut-off date, we think that this is not a problem. To our knowledge, we are the first who try to fuzz applications using a LLM analysis based on CFGs. Furthermore, it is impractical to filter on bugs who were discovered after the cut-off date. The cut-off date is quite recent, and as explained in Section 1.1, the number of available fuzzing benchmarks is low. Besides the low number of available fuzzing benchmarks, LLMs are in a rapidly changing field, making it unrealistic to find a LLM model that is both state of the art and has a cut-off date far in the past. Therefore, it would be impossible to find new CVEs ourselves and create a benchmark based on the found bugs.

We do see that there is a difference between the models, as can be seen in Table 4 and 5. Although the differences are very minimal, LLama 4 Maverick can reach and trigger the most bugs. Furthermore, an interesting result is the result with GPT-4.1. GPT-4.1 is a state-of-the-art model. With this, the cost to use the model is higher, and it takes more time to generate an answer. Nonetheless, GPT-4.1 is the model who is the quickest in reaching the `PNG006` bug. We do see that the Meta LLama 4 model identifies these bugs later, but can also trigger `PNG002`. One explanation for this can be that, since the cost to use Meta LLama 4 is lower, we have more room to improve the generated code. Furthermore, we can switch more between which function we want to target, increasing our chances that we find an entry that can reach or trigger vulnerable code.
Additionally, we see that there is no clear distinction between the costs of generating the CFGs and the fuzzing process. The amount of money used to analyze the CFGs differs a lot per model and target. Therefore, we cannot answer SR5.

The ablation experiment without a custom mutator shows that the use of a custom mutator is actually worsening the results of the fuzzing campaign. Without the custom mutator, we see that we can reach and trigger a bug more than with a custom mutator. This shows that, at least with this particular target, the custom mutator has not the expected effect on the fuzzing campaign. We can answer sub-question SR6 with the answer that the custom mutator seems to have a negative effect on the efficiency of the generated fuzzing harness with the Control Flow Graph analysis.

With all the information gathered from answering the sub-research questions, we can answer our research question 'How can we automate the fuzzing process of existing C-style programs, and generate custom mutators, fuzzing harnesses, and seed input files?'. The different experiments show that, depending on the model, LLM2Fuzz is capable of automating the fuzzing process by automatically writing fuzzing harnesses, custom mutators and seed files. We automated these steps by using Large Language Models to generate the required files. The automating process can further be improved by dynamically testing the fuzzing harness before we start creating seed input files. We do see a small improvement of the distinct number of bugs reached against human written and by OSS-Fuzz-Gen generated harnesses. As our ablation study shows, we see that the Control Flow Graph analysis seems to have a positive effect on the time it takes to reach a bug, while the use of custom mutators seems to have a negative effect on the distinct number of bugs found. This does seem to show that the use of the Control Flow Graph analysis has a positive effect on the efficiency, while the use of custom mutators does have a negative effect on the efficiency. However, more runs with

LLM2Fuzz are needed to prove the effectiveness of the tool.

In conclusion, we see that LLM2Fuzz is capable of automating the fuzzing process, by generating fuzzing harness, custom mutators and seed input files. Furthermore, the CFG analyzed, LLM generated harnesses and custom mutators seem to perform a bit better than humanly written harnesses and current state of the art solutions. In addition, the implementation seems to show a positive effect on the time it takes to reach the first bug. Nonetheless, the differences are slim, and future research using larger LLMs or dynamically tested harnesses are needed to verify the impact of the CFG analysis on the generation of fuzzing harnesses and custom mutators.

Appendixes

Two appendixes are added to the thesis. Appendix A contains all the used prompts, and Appendix B contains all the abbreviations.

A Prompts

The following prompts are used in the implementation.

A.1 CFG generation

The following prompts are used for the CFG generation.

A.1.1 JSON format example

```
JSON_CFG_EXAMPLE = """
{
  "explanation": "explanation in string form here",
  "functions": [
    {
      "name": "example_function_1",
      "parameters": [
        {
          "type": "i32",
          "detail: "Example detail"
        }
      ],
      "vulnerabilityRating": 30,
      "explanation": "explanation why the function is vulnerable",
      "pathsToFunction": [
        "main",
        "function2"
      ]
    }
    ]
}

"""
```

A.1.2 CFG generation prompt

```
cfg_generation_instruction_prompt = (
    " You are prompted with a CFG in the DOT-language generated by Clang/LLVM.
        The programs are written in C/C++. It may be possible that you get
      multiple CFGs, f.e. for different functions used in the code. We start
       every CFG with <CFG_index> and end the CFG with </CFG_index>, where
      index is a number. We want to use these CFGs for guided-fuzzing. Your
      job is to list all functions that can be fuzzed, so we can prompt
      another chat to generate the fuzzing harnasses. Make a list of"
    " all the functions that can be fuzzed and give some details about the
      parameters. Furthermore, examine the code to see which function would
      be most prone to vulnerable bugs, so we can fuzz these function first.
       Make a list"
```

```
    " of these functions and give them a rating from 0-100% on how vulnerable
        you think they are. At last, describe al the"
    " different paths the code execution can take to get to that function.
        List ALL the different and ALL the paths the program should take to
        get to the vulnerable functions; use the REAL function names, NOT the
        addresses."
    " After the initial prompt, you can be asked to redo the analysis. Use the
         same CFGs as the first prompt, but start with your analysis from 0."
    f" Return the response in JSON the following format: {JSON_CFG_EXAMPLE}.
        ONLY return the JSON, do NOT include the ```json ``` tags: THIS IS
        VERY IMPORTANT. Please do NOT include any standard functions, so no
        functions that are in the standard library (STD), iterators or
        anything else. List ALL the functions you found in the CFG/in the code
        ; even in the rating is low."
)
```

A.2 Mutator and Harness generation

The mutator example, harness example and mutator documentation can be found in the code. (note/question: these are very long prompts, and not very intersting. Should I include them here?)

A.2.1 Generation prompt

```
    mutator_generation_instruction_prompt = (
    "You are a cyber security testing engineer with the task to write a AFL++
        fuzzing harness and custom mutator. Your task is to write these custom
         mutators and harnesses using Control Flow Graphs."
    "Your job is to use this domain-specific information, the analysis based
        on Control Flow Graphs, to create a harness and custom mutator which
        targets all the different code paths "
    "in the function you are assigned to target. Make sure to ONLY target this
         function, and not any other. Make really sure to use the anaysis for
        this task."
    " For this, you are given the following information: "
    "1. The analysis done on the Control Flow Graphs by a previous AI agent.
        In here, you see which functions are probably the most vulnerable and
        which you should target first. This is given between <ANALYSIS> and </
        ANALYSIS> tags. Please adhere to this information. "
    "2. The source code of the program. This is given to you between <
        CODE_FILENAME> and </CODE_FILENAME>. Header files are given to you by
        using <CODEH_FILENAME> and </CODEH_FILENAME>, where FILENAME is the
        name of the file. Use this information to include the right files in
        the fuzzer! "
    "3. The function we want to target. This is given to you between <FUNC>
        and </FUNC>"
    "4. The output of 'ls' in <LS> and </LS>. Here you can find the right file
         names to include."
    "Make sure to identify which files you have to include, and get the
        included files right!"

    f"Avoid hardcoding strings; we do want to reach all the paths, but try to
        create a generic mutator for this. The following functions can be used
         for the mutator: <code> {mut_functions} </code>, with the following
```

```
        documentation: <documentation> {mut_documentation} </documentation>. A
         mutator example can be found here: <code> {mutator_example} </code>"
    "Also, do NOT create weak links for functions. We can NOT fuzz using this
        approach. Use the FILENAMES to INCLUDE (using #include) the right
        files! Does including the .h file not work? Try including the .c file
        ."
    "We expect the following output, in JSON: "
    "1. The source of the harness, with \"harness\" as tag. Do use the
        LLVMFuzzerTestOneInput function instead of main."
    "2. The custom mutator, with \"custom_mutator\" as tag."
    "3. The harness file extension of the file (.c or .cpp), with \"
        harness_extension\" as tag."
    "4. The custom mutator file extension (.c or .cpp) of the file, with \"
        mutator_extension\" as tag."
    "5. A small explanation, with \"explanation\" as tag"
    "Make sure to adhere to this output, and do not add anything else."

    "After the first prompt, we can ask you one of the following things:"
    "1. Create a custom mutator and harness to reach another function. "
    "2. Fix the current harness or mutator. We will give you the compiler
        error."
    "3. Improve the current custom mutator and harness. We will prompt you
        with a LCOV Trace File and the number of unique crashes. Try to
        improve or write a new current harness to obtain better coverage and,
        more importantly, find new crashes. You should say so if you think if
        the custom"
    "mutator and harness are already on it's best, so we can continue to the
        next function. Try to avoid improving the current custom mutator or
        harness; we want to continue to the next function in reasonable time.
        Please remember that a higher number of crashes means that the fuzzing
         is better."
)
```

A.3 Seed generation

A.3.1 JSON format example

```
JSON_SEEDS_EXAMPLE = """
{
  "files": [
    {
      "file": "filename_1",
      "content": "content file 1",
      "explanation": "small explanation of the file"
    },
    {
      "file": "filename_2",
      "content": "content file 2",
      "explanation": "small explanation of the file",
    }
    ]
}
"""
```

A.3.2 Generation prompt

```
seeds_generation_prompt = (
    "You are a cyber security testing engineer who wants to create input files
        which can be used to fuzz a program with AFL++. Your job is to create
        input files, whereby each input file will take another code path in
       the function we want to fuzz. Make sure that we take ALL the different
        paths in the function we want to fuzz. "
    "Make sure that these inputs do NOT crash; that is a requirement of AFL++.
        Double check this requirement with the code; perform static analysis
       to achieve this! Explore as much code lines as possible for the
       function we provide, but again, let the program NOT CRASH. "
    "Check your seeds with the Control Flow Graphs (CFGs) that are returned to
        ensure that all possible, non-crashing paths are taken. Please do a
       deep analysis on the source code and CFGs which are presented to you,
       "
    f"and respond according to the following format: {JSON_SEEDS_EXAMPLE}. Do
       NOT add the ```json' / ````' tags. Always, no matter what, adhere to
       this format. \n You will be prompted as follows: \n <CFGS> CGS HERE </
       CFGS> <CODE> CODE HERE </CODE> \n <FUNC> FUNCTION TO WRITE THE INPUT
       FILES FOR </FUNC>."
    "It may be the case that we will ask you to fix certain files, f.e. if
       they still crash. If we do that, come up with completely new seed
       files. "
)
```

B Abbreviations

In this research, multiple abbreviations are used. This list contains all full-written abbreviations in alphabetic order.

- AFL - American Fuzzy Lop
- AFL++ - American Fuzzy Lop plus plus
- AI - Artificial Intelligence
- CAPEC - Common Attack Pattern Enumerations and Classifications
- CFG - Control Flow Graph
- CVE - Common Vulnerabilities and Exposures
- CVSS - Common Vulnerability Scoring System
- GDB - GNU Debugger
- LLM - Large Language Model
- PNG - Portable Network Graphics
- TNO - Nederlandse organisatie voor toegepast-natuurwetenschappelijk onderzoek
- QEMU - QuickEmulator

References

[1] Corpus - The LibAFL Fuzzing Library — aflplus.plus. https://aflplus.plus/libafl-book/core_concepts/corpus.html.

[2] Coverage guided vs blackbox fuzzing — google.github.io. https://google.github.io/clusterfuzz/reference/coverage-guided-vs-blackbox/. [Accessed 25-07-2025].

[3] Custom Mutators — AFLplusplus — aflplus.plus. https://aflplus.plus/docs/custom_mutators/.

[4] Fuzzing — appsec.guide. https://appsec.guide/docs/fuzzing/.

[5] Libfuzzer: A library for coverage-guided fuzz testing. URL: https://llvm.org/docs/LibFuzzer.html.

[6] OpenRouter — openrouter.ai. URL: openrouter.ai.

[7] Wuppiefuzz v1.2.0. URL: https://github.com/TNO-S3/WuppieFuzz.

[8] AFL++. American Fuzzy Lop Plus Plus (afl++). URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/README.md.

[9] AFL++. Custom mutators in afl++. URL: https://aflplus.plus/docs/custom_mutators/.

[10] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery. doi:10.1145/800028.808479.

[11] Craig Beaman, Michael Redbourne, J. Darren Mummery, and Saqib Hakak. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers Security*, 120:102813, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002073, doi:10.1016/j.cose.2022.102813.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[13] Corinne Bernstein. What is happy path testing?: Definition from techtarget, Oct 2024. [Accessed 25-07-2025]. URL: https://www.techtarget.com/searchsoftwarequality/definition/happy-path-testing.

[14] Kira Bobrovnikova, Sergii Lysenko, Bohdan Savenko, Piotr Gaj, and Oleg Savenko. Technique for iot malware detection based on control flow graph analysis. *Radioelectronic and Computer Systems*, (1):141–153, 2022.

[15] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. {WhisperFuzz}:{White-Box} fuzzing for detecting and locating timing vulnerabilities in processors. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5377–5394, 2024.

[16] Thomas Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083, March 1997. URL: https://www.rfc-editor.org/info/rfc2083, doi:10.17487/RFC2083.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[18] CanaryTrap. Black-Box, White-Box, and Gray-Box testing explained. URL: https://www.canarytrap.com/blog/black-box-white-box-and-gray-box-testing-explained/.

[19] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016. doi:10.1109/SP.2016.15.

[20] Mohamed Amine Ferrag, Fatima Alwahedi, Ammar Battah, Bilel Cherif, Abdechakour Mechri, and Norbert Tihanyi. Generative ai and large language models for cyber security: All insights you need. *Available at SSRN 4853709*, 2024.

[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[22] Jonathan Foote. GDB 'exploitable' plugin, 2015. URL: `https://github.com/jfoote/exploitable`.

[23] Python Software Foundation. queue — a synchronized queue class.

[24] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[25] Google. American Fuzzy Lop. URL: `https://github.com/google/AFL`.

[26] Graphviz. Dot language, 2024. URL: `https://graphviz.org/doc/info/lang.html`.

[27] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020. `doi:10.1145/3428334`.

[28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark, 2020. URL: `https://hexhive.epfl.ch/magma/`.

[29] Jie Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai, 2023. URL: `https://arxiv.org/abs/2306.06782`, `arXiv:2306.06782`.

[30] mbarbella-chromium kcc, morehouse. What makes a good fuzz target. URL: `https://github.com/google/fuzzing/blob/master/docs/good-fuzz-target.md`.

[31] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., USA, 1978.

[32] Hans Sjouke Koopal. Cybersecurity, May 2025. URL: `https://www.abnamro.com/en/news/one-in-five-dutch-companies-harmed-by-cyberattacks-in-2024`.

[33] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[34] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[35] Abhishek Arya Oliver Chang Jonathan Metzman Kostya Serebryany Dongge Liu. Oss-fuzz. (Accessed: 13-11-2024). URL: `https://github.com/google/oss-fuzz`.

[36] Sanoop Mallissery and Yu-Sung Wu. Demystify the fuzzing methods: A comprehensive survey. *ACM Computing Surveys*, 56(3):1–38, 2023.

[37] Shaswata Mitra, Stephen A Torri, and Sudip Mittal. Survey of malware analysis through control flow graph using machine learning. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1554–1561. IEEE, 2023.

[38] mrash. afl-cov - AFL Fuzzing Code Coverage. URL: `https://github.com/mrash/afl-cov`.

[39] Tjada Nelson. Malware classification using weighted control flow graphs. 2025.

[40] OpenAI. Structured outputs. URL: `https://platform.openai.com/docs/guides/structured-outputs?api-mode=chat`.

[41] OpenAI. Introducing ChatGPT, 2022. URL: https://openai.com/index/chatgpt/.

[42] ossf. Fuzz-introspect. URL: https://github.com/ossf/fuzz-introspector.

[43] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018. doi:10.1109/SP.2018.00056.

[44] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. Fine-grained code coverage measurement in automated black-box android testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–35, 2020.

[45] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[46] Radare2 Team. *Radare2 Book*. GitHub, 2017.

[47] wcl-sayfer admin. Fuzzing Part 1: The Theory - Sayfer — sayfer.io. https://sayfer.io/blog/fuzzing-part-1-the-theory/.

[48] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3597503.3639121.

[49] Hang Xu, Liheng Chen, Shuitao Gan, Chao Zhang, Zheming Li, Jiangan Ji, Baojian Chen, and Fan Hu. Graphuzz: Data-driven seed scheduling for coverage-guided greybox fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 33(7), August 2024. doi:10.1145/3664603.

[50] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. Large language models for cyber security: A systematic literature review, 2025. URL: https://arxiv.org/abs/2405.04760, arXiv:2405.04760.

[51] Muhammad Mudassar Yamin, Ehtesham Hashmi, Mohib Ullah, and Basel Katt. Applications of llms for generating cyber security exercise scenarios. *IEEE Access*, 2024.

[52] Jingda Yang, Sudhanshu Arya, and Ying Wang. Formal-guided fuzz testing: Targeting security assurance from specification to implementation for 5g and beyond. *IEEE Access*, 12:29175–29193, 2024.

[53] He Yuan, Xue Bo, Zhang Lina, and Lu Chengyang. A control flow graph optimization method for enhancing fuzz testing. *IEEE Access*, 12:169370–169378, 2024. doi:10.1109/ACCESS.2024.3452938.

[54] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of embedded systems: A survey. *ACM Computing Surveys*, 55(7):1–33, 2022.

[55] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Search-based fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-11-11 18:18:06+01:00. URL: https://www.fuzzingbook.org/html/SearchBasedFuzzer.html.

[56] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Mutation-based fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-11-09 17:25:56+01:00. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html.

[57] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. Llamafuzz: Large language model enhanced greybox fuzzing, 2024. URL: https://arxiv.org/abs/2406.07714, arXiv:2406.07714.

[58] Jie Zhang, Haoyu Bu, Hui Wen, Yongji Liu, Haiqiang Fei, Rongrong Xi, Lun Li, Yun Yang, Hongsong Zhu, and Dan Meng. When llms meet cybersecurity: A systematic literature review. *Cybersecurity*, 8(1):55, 2025.

[59] Minmin Zhou, Jinfu Chen, Yisong Liu, Hilary Ackah-Arthur, Shujie Chen, Qingchen Zhang, and Zhifeng Zeng. A method for software vulnerability detection based on improved control flow graph. *Wuhan University Journal of Natural Sciences*, 24(2):149–160, Apr 2019. doi:10.1007/s11859-019-1380-z.

[60] Zhengxiang Zhou and Cong Wang. Shadowbug: Enhanced synthetic fuzzing benchmark generation. *IEEE Open Journal of the Computer Society*, 2024.

[61] Kailong Zhu, Yuliang Lu, Hui Huang, Lu Yu, and Jiazhen Zhao. Constructing more complete control flow graphs utilizing directed gray-box fuzzing. *Applied Sciences*, 11(3):1351, 2021.

[62] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.